

```

*****
47494 Thu Feb 25 15:39:32 2016
new/usr/src/cmd/fm/fmadm/common/faulty.c
2976 remove useless offsetof() macros
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
23 */

25 #include <sys/types.h>
26 #include <fmadm.h>
27 #include <errno.h>
28 #include <limits.h>
29 #include <strings.h>
30 #include <stdio.h>
31 #include <unistd.h>
32 #include <sys/wait.h>
33 #include <sys/stat.h>
34 #include <fcntl.h>
35 #include <fm/fmd_log.h>
36 #include <sys/fm/protocol.h>
37 #include <fm/libtopo.h>
38 #include <fm/fmd_adm.h>
39 #include <fm/fmd_msg.h>
40 #include <dlfcn.h>
41 #include <sys/systeminfo.h>
42 #include <sys/utsname.h>
43 #include <libintl.h>
44 #include <locale.h>
45 #include <sys/smbios.h>
46 #include <libdevinfo.h>
47 #include <stdlib.h>
48 #include <stddef.h>

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

50 /*
51 * Fault records are added to catalog by calling add_fault_record_to_catalog()
52 * records are stored in order of importance to the system.
53 * If -g flag is set or not_suppressed is not set and the class fru, fault,
54 * type are the same then details are merged into an existing record, with uuid
55 * records are stored in time order.
56 * For each record information is extracted from nvlist and merged into linked
57 * list each is checked for identical records for which percentage certainty are
58 * added together.
59 * print_catalog() is called to print out catalog and release external resources

```

```

60 *
61 *
62 * status_rec_list -> {-----} -|
63 *
64 *
65 *
66 * status_fru_list {-----} -> {-----} -> {-----} -|
67 *                  |         |         |
68 *                  |         |         |
69 *                  |         |         |
70 *                  |         |         |
71 *                  |         |         |
72 *                  |         |         |
73 *                  |         |         |
74 * status_asru_list {-----} -> {-----} -> {-----} -|
75 *                  |         |         |
76 *                  |         |         |
77 *                  |         |         |
78 *                  |         |         |
79 *                  |         |         |
80 *                  |         |         |
81 *                  |         |         |
82 *                  |         |         |
83 *                  |         |         |
84 *
85 * Fmadm faulty takes a number of options which affect the format of the
86 * output displayed. By default, the display reports the FRU and ASRU along
87 * with other information on per-case basis as in the example below.
88 *
89 * -----
90 * TIME                EVENT-ID                MSG-ID                SEVERITY
91 * -----
92 * Sep 21 10:01:36    d482f935-5c8f-e9ab-9f25-d0aaafec1e6c    AMD-8000-2F          Major
93 *
94 * Fault class       : fault.memory.dimmb
95 * Affects           : mem:///motherboard=0/chip=0/memory-controller=0/dimm=0/rank=0
96 *                   : faulted but still in service
97 * FRU                : "CPU 0 DIMM 0" (hc:///.../memory-controller=0/dimm=0)
98 *                   : faulty
99 *
100 * Description       : The number of errors associated with this memory module has
101 *                   : exceeded acceptable levels. Refer to
102 *                   : http://illumos.org/msg/AMD-8000-2F for more information.
103 *
104 * Response          : Pages of memory associated with this memory module are being
105 *                   : removed from service as errors are reported.
106 *
107 * Impact            : Total system memory capacity will be reduced as pages are
108 *                   : retired.
109 *
110 * Action            : Schedule a repair procedure to replace the affected memory
111 *                   : module. Use fmdump -v -u <EVENT_ID> to identify the module.
112 *
113 * The -v flag is similar, but adds some additional information such as the
114 * resource. The -s flag is also similar but just gives the top line summary.
115 * All these options (ie without the -f or -r flags) use the print_catalog()
116 * function to do the display.
117 *
118 * The -f flag changes the output so that it appears sorted on a per-fru basis.
119 * The output is somewhat cut down compared to the default output. If -f is
120 * used, then print_fru() is used to print the output.
121 *
122 * -----
123 * "SLOT 2" (hc:///.../hostbridge=3/pciexrc=3/pciexbus=4/pciexdev=0) faulty
124 * 5ca4aeb3-36...f6be-c2e8166dc484 2 suspects in this FRU total certainty 100%
125 *

```

```
126 * Description : A problem was detected for a PCI device.
127 *             Refer to http://illumos.org/msg/PCI-8000-7J
128 *             for more information.
129 *
130 * Response    : One or more device instances may be disabled
131 *
132 * Impact      : Possible loss of services provided by the device instances
133 *             associated with this fault
134 *
135 * Action      : Schedule a repair procedure to replace the affected device.
136 *             Use fmdump -v -u <EVENT_ID> to identify the device or contact
137 *             Sun for support.
138 *
139 * The -r flag changes the output so that it appears sorted on a per-asru basis.
140 * The output is very much cut down compared to the default output, just giving
141 * the asru fmri and state. Here print_asru() is used to print the output.
142 *
143 * mem:///motherboard=0/chip=0/memory-controller=0/dimm=0/rank=0          degraded
144 *
145 * For all fmadm faulty options, the sequence of events is
146 *
147 * 1) Walk through all the cases in the system using fmd_adm_case_iter() and
148 * for each case call dfault_rec(). This will call add_fault_record_to_catalog()
149 * This will extract the data from the nvlist and call catalog_new_record() to
150 * save the data away in various linked lists in the catalogue.
151 *
152 * 2) Once this is done, the data can be supplemented by using
153 * fmd_adm_rsrc_iter(). However this is now only necessary for the -i option.
154 *
155 * 3) Finally print_catalog(), print_fru() or print_asru() are called as
156 * appropriate to display the information from the catalogue sorted in the
157 * requested way.
158 *
159 */

161 typedef struct name_list {
162     struct name_list *next;
163     struct name_list *prev;
164     char *name;
165     uint8_t pct;
166     uint8_t max_pct;
167     ushort_t count;
168     int status;
169     char *label;
170 } name_list_t;
unchanged portion omitted
```

new/usr/src/cmd/mdb/common/modules/ii/ii.c

1

```
*****
12270 Thu Feb 25 15:39:33 2016
new/usr/src/cmd/mdb/common/modules/ii/ii.c
2976 remove useless offsetof() macros
*****
```

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */
```

```
26 #include <stddef.h>
```

```
28 #endif /* ! codereview */
```

```
29 #include <sys/types.h>
```

```
30 #include <sys/mdb_modapi.h>
```

```
32 #include <sys/nsctl/nsctl.h>
```

```
33 #include <sys/unistat/spcs_s.h>
```

```
34 #include <sys/unistat/spcs_s_k.h>
```

```
37 #include <sys/nsctl/dsw.h>
```

```
38 #include <sys/nsctl/dsw_dev.h>
```

```
40 #include <sys/nsctl/nsvers.h>
```

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

```
43 const mdb_bitmask_t bi_flags_bits[] = {
44     { "DSW_GOLDEN", DSW_GOLDEN, DSW_GOLDEN },
45     { "DSW_COPYINGP", DSW_COPYINGP, DSW_COPYINGP },
46     { "DSW_COPYINGM", DSW_COPYINGM, DSW_COPYINGM },
47     { "DSW_COPYINGS", DSW_COPYINGS, DSW_COPYINGS },
48     { "DSW_COPYINGX", DSW_COPYINGX, DSW_COPYINGX },
49     { "DSW_BMPOFFLINE", DSW_BMPOFFLINE, DSW_BMPOFFLINE },
50     { "DSW_SHDOFFLINE", DSW_SHDOFFLINE, DSW_SHDOFFLINE },
51     { "DSW_MSTOFFLINE", DSW_MSTOFFLINE, DSW_MSTOFFLINE },
52     { "DSW_OVROFFLINE", DSW_OVROFFLINE, DSW_OVROFFLINE },
53     { "DSW_TREEMAP", DSW_TREEMAP, DSW_TREEMAP },
54     { "DSW_OVERFLOW", DSW_OVERFLOW, DSW_OVERFLOW },
55     { "DSW_SHDEXPORT", DSW_SHDEXPORT, DSW_SHDEXPORT },
56     { "DSW_SHDIMPORT", DSW_SHDIMPORT, DSW_SHDIMPORT },
57     { "DSW_VOVERFLOW", DSW_VOVERFLOW, DSW_VOVERFLOW },
58     { "DSW_HANGING", DSW_HANGING, DSW_HANGING },
59     { "DSW_CFGOFFLINE", DSW_CFGOFFLINE, DSW_CFGOFFLINE },
```

new/usr/src/cmd/mdb/common/modules/ii/ii.c

2

```
60     { "DSW_OVRHDRDRTY", DSW_OVRHDRDRTY, DSW_OVRHDRDRTY },
61     { "DSW_RESIZED", DSW_RESIZED, DSW_RESIZED },
62     { "DSW_FRECLAIM", DSW_FRECLAIM, DSW_FRECLAIM },
63     { NULL, 0, 0 }
64 };
```

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

```

1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

```

```

26 #ifndef _MDBMOD_MISC_H
27 #define _MDBMOD_MISC_H

```

```

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

```

```

29 #include <mdb/mdb_modapi.h>

```

```

30 #include <stddef.h>

```

```

31 #endif /* ! codereview */

```

```

33 #ifdef __cplusplus

```

```

34 extern "C" {

```

```

35 #endif

```

```

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

```

```

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

```

```

39 extern int umem_set_standalone(void);

```

```

40 extern ssize_t umem_lookup_by_name(const char *, GElf_Sym *);

```

```

41 extern ssize_t umem_readvar(void *, const char *);

```

```

43 /*

```

```

44  * Returns non-zero if sym matches libumem*'prefix'

```

```

45  */

```

```

46 int is_umem_sym(const char *, const char *);

```

```

48 #define dprintf(x) if (umem_debug_level) { \

```

```

49     mdb_printf("umem debug: "); \

```

```

50     /*CSTYLED*\

```

```

51     mdb_printf x ;\

```

```

52 }

```

_____unchanged_portion_omitted_____

```

*****
27115 Thu Feb 25 15:39:34 2016
new/usr/src/cmd/pools/poolstat/poolstat.c
2976 remove useless offsetof() macros
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 /*
27  * poolstat - report active pool statistics
28  */
29 #include <stdio.h>
30 #include <unistd.h>
31 #include <stdlib.h>
32 #include <unistd.h>
33 #include <locale.h>
34 #include <string.h>
35 #include <ctype.h>
36 #include <limits.h>
37 #include <errno.h>
38 #include <stddef.h>
39 #endif /* ! codereview */

41 #include <pool.h>
42 #include "utils.h"
43 #include "poolstat.h"
44 #include "poolstat_utils.h"
45 #include "statcommon.h"

47 #ifndef TEXT_DOMAIN
48 #define TEXT_DOMAIN "SYS_TEST"
49 #endif

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

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

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

59 static uint_t timestamp_fmt = NODATE;

```

```

61 /* available field formatters */
62 static int default_f(char *, int, int, poolstat_field_format_t *, char *);
63 static int bigno_f(char *, int, int, poolstat_field_format_t *, char *);
64 static int used_stat_f(char *, int, int, poolstat_field_format_t *, char *);
65 static int header_f(char *, int, int, poolstat_field_format_t *, char *);

67 /* statistics bags used to collect data from various provider */
68 static statistic_bag_t pool_sb_ag_s;
69 static statistic_bag_t pset_sb_ag_s;
70 static statistic_bag_t *pool_sb_ag = &pool_sb_ag_s;
71 static statistic_bag_t *pset_sb_ag = &pset_sb_ag_s;

73 /* formatter objects for pset, defined in a default printing sequence */
74 static poolstat_field_format_t pset_ffs[] = {
75     /* prt flags,name,header,type,width,minwidth,offset,formatter */
76     { DX_FIELD, "id", "id", LL, 3, 1, offsetof(pool_sb_ag),
77       (formatter)default_f },
78     { DX_FIELD, "pool", "pool", STR, 20, 14, offsetof(pool_sb_ag),
79       (formatter)default_f },
80     { DX_FIELD, "type", "type", STR, 4, 5, offsetof(pset_sb_ag),
81       (formatter)default_f },
82     { D_FIELD, "rid", "rid", LL, 3, 1, offsetof(pset_sb_ag_s.bag),
83       (formatter)default_f },
84     { DX_FIELD, "rset", "rset", STR, 20, 14, offsetof(pset_sb_ag),
85       (formatter)default_f },
86     { DX_FIELD, "min", "min", ULL, 4, 1, offsetof(pset_sb_ag_s.bag),
87       (formatter)bigno_f },
88     { DX_FIELD, "max", "max", ULL, 4, 1, offsetof(pset_sb_ag_s.bag),
89       (formatter)bigno_f },
90     { DX_FIELD, "size", "size", ULL, 4, 1, offsetof(pset_sb_ag_s.bag),
91       (formatter)default_f },
92     { DX_FIELD, "used", "used", FL, 4, -1, offsetof(pset_sb_ag_s.bag),
93       (formatter)used_stat_f },
94     { DX_FIELD, "load", "load", FL, 4, -1, offsetof(pset_sb_ag_s.bag),
95       (formatter)default_f }
96 };

```

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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 2004, 2010, Oracle and/or its affiliates. All rights reserved.
23 */

25 #include "statcommon.h"
26 #include "dsr.h"

28 #include <sys/dklabel.h>
29 #include <sys/dktp/fdisk.h>
30 #include <stdlib.h>
31 #include <stdarg.h>
32 #include <stddef.h>
33 #endif /* ! codereview */
34 #include <unistd.h>
35 #include <strings.h>
36 #include <errno.h>
37 #include <limits.h>

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

41 static struct iodev_snapshot *
42 make_controller(int cid)
43 {
44     struct iodev_snapshot *new;

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

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

54     return (new);
55 }

57 static struct iodev_snapshot *
58 find_iodev_by_name(struct iodev_snapshot *list, const char *name)
59 {
60     struct iodev_snapshot *pos;
61     struct iodev_snapshot *pos2;

```

```

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

67         pos2 = find_iodev_by_name(pos->is_children, name);
68         if (pos2 != NULL)
69             return (pos2);
70     }

72     return (NULL);
73 }

75 static enum iodev_type
76 parent_iodev_type(enum iodev_type type)
77 {
78     switch (type) {
79         case IODEV_CONTROLLER: return (0);
80         case IODEV_IOPATH_LT: return (0);
81         case IODEV_IOPATH_LI: return (0);
82         case IODEV_NFS: return (0);
83         case IODEV_TAPE: return (0);
84         case IODEV_IOPATH_LTI: return (IODEV_DISK);
85         case IODEV_DISK: return (IODEV_CONTROLLER);
86         case IODEV_PARTITION: return (IODEV_DISK);
87     }
88     return (IODEV_UNKNOWN);
89 }

91 static int
92 id_match(struct iodev_id *id1, struct iodev_id *id2)
93 {
94     return (id1->id == id2->id &&
95         strcmp(id1->tid, id2->tid) == 0);
96 }

98 static struct iodev_snapshot *
99 find_parent(struct snapshot *ss, struct iodev_snapshot *iodev)
100 {
101     enum iodev_type parent_type = parent_iodev_type(iodev->is_type);
102     struct iodev_snapshot *pos;
103     struct iodev_snapshot *pos2;

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

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

112     if (parent_type == IODEV_CONTROLLER) {
113         for (pos = ss->s_iodevs; pos; pos = pos->is_next) {
114             if (pos->is_type != IODEV_CONTROLLER)
115                 continue;
116             if (pos->is_id.id != iodev->is_parent_id.id)
117                 continue;
118             return (pos);
119         }
121     if (!(ss->s_types & SNAP_CONTROLLERS))
122         return (NULL);

124     pos = make_controller(iodev->is_parent_id.id);
125     insert_iodev(ss, pos);
126     return (pos);
127 }

```

```
129     /* IODEV_DISK parent */
130     for (pos = ss->s_io devs; pos; pos = pos->is_next) {
131         if (id_match(&iodev->is_parent_id, &pos->is_id) &&
132             pos->is_type == IODEV_DISK)
133             return (pos);
134         if (pos->is_type != IODEV_CONTROLLER)
135             continue;
136         for (pos2 = pos->is_children; pos2; pos2 = pos2->is_next) {
137             if (pos2->is_type != IODEV_DISK)
138                 continue;
139             if (id_match(&iodev->is_parent_id, &pos2->is_id))
140                 return (pos2);
141         }
142     }
144     return (NULL);
145 }

147 /*
148  * Introduce an index into the list to speed up insert_into looking for the
149  * right position in the list. This index is an AVL tree of all the
150  * iodev_snapshot in the list.
151  */

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

152 static int
153 avl_iodev_cmp(const void* is1, const void* is2)
154 {
155     int c = iodev_cmp((struct iodev_snapshot *)is1,
156                     (struct iodev_snapshot *)is2);
158     if (c > 0)
159         return (1);
161     if (c < 0)
162         return (-1);
164     return (0);
165 }
unchanged_portion_omitted
```

```

*****
75112 Thu Feb 25 15:39:35 2016
new/usr/src/common/nvpair/nvpair.c
2976 remove useless offsetof() macros
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
22 /*
23  * Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 #include <sys/stropts.h>
27 #include <sys/debug.h>
28 #include <sys/isa_defs.h>
29 #include <sys/int_limits.h>
30 #include <sys/nvpair.h>
31 #include <sys/nvpair_impl.h>
32 #include <rpc/types.h>
33 #include <rpc/xdr.h>

35 #if defined(_KERNEL) && !defined(_BOOT)
36 #include <sys/varargs.h>
37 #include <sys/ddi.h>
38 #include <sys/sunddi.h>
39 #include <sys/sysmacros.h>
40 #endif /* !codereview */
41 #else
42 #include <stdarg.h>
43 #include <stdlib.h>
44 #include <string.h>
45 #include <strings.h>
46 #include <stddef.h>
47 #endif /* !codereview */
48 #endif

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

52 /*
53  * nvpair.c - Provides kernel & userland interfaces for manipulating
54  * name-value pairs.
55  *
56  * Overview Diagram
57  *
58  * +-----+

```

```

59 * | nvlist_t
60 * |-----|
61 * | nvl_version
62 * | nvl_nvflag
63 * | nvl_priv  +-+
64 * | nvl_flag
65 * | nvl_pad
66 * |-----+
67 * |
68 * |
69 * |
70 * |
71 * |
72 * |
73 * |
74 * |
75 * |
76 * |
77 * |
78 * |
79 * |
80 * |
81 * |
82 * |
83 * |
84 * |
85 * |
86 * |
87 * |
88 * |
89 * |
90 * |
91 * |
92 * |
93 * |
94 * |
95 * |
96 * |
97 * |
98 * |
99 * |
100 * |
101 * |
102 * |
103 * |
104 * |
105 * |
106 * |
107 * |
108 * |
109 * |
110 * |
111 * |
112 * |
113 * |
114 * |
115 * |
116 * |
117 * |
118 * |
119 * |
120 * |
121 * |
122 * |
123 * |
124 * |

```

Diagram illustrating the structure of nvlist_t and its components:

- nvlist_t** (lines 59-65): Contains nvl_version, nvl_nvflag, nvl_priv, nvl_flag, and nvl_pad.
- nvpriv_t** (lines 69-74): Contains nvp_list, nvp_last, nvp_curr, nvp_nva, and nvp_stat.
- nv_alloc_t** (lines 72-77): Contains nva_ops and nva_arg.
- i_nvpair_t** (lines 81-91): Contains nvi_next, nvi_prev, nvp (nvpair_t), nvp_size, nvp_name_sz, nvp_value_elem, nvp_type, and data.
- i_nvpair_t (last)** (lines 96-108): Contains nvi_next (NULL), nvi_prev, nvp (nvpair_t), nvp_size, nvp_name_sz, nvp_value_elem, nvp_type, and data.
- nvlist_t** (lines 112-117): Contains nvl_version, nvl_nvflag, nvl_priv, nvl_flag, and nvl_pad.

Additional notes:

- Line 121: * N.B. nvpair_t may be aligned on 4 byte boundary, so +4 will
- Line 122: * allow value to be aligned on 8 byte boundary
- Line 124: * name_len is the length of the name string including the null terminator


```
125 * so it must be >= 1
126 */
127 #define NVP_SIZE_CALC(name_len, data_len) \
128     (NV_ALIGN(sizeof (nvpair_t)) + name_len) + NV_ALIGN(data_len)

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

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

138 #define NVP_VALOFF(nvp) (NV_ALIGN(sizeof (nvpair_t) + (nvp)->nvp_name_sz))
139 #define NVP2I_NVP(nvp) \
140     ((i_nvp_t *) ((size_t) (nvp) - offsetof(i_nvp_t, nvi_nvp)))

143 int
144 nv_alloc_init(nv_alloc_t *nva, const nv_alloc_ops_t *nvo, /* args */ ...)
145 {
146     va_list valist;
147     int err = 0;

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

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

157     return (err);
158 }
unchanged_portion_omitted_
```

new/usr/src/head/iso/stddef_iso.h

1

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

83 #if __GNUC__ > 4 || (__GNUC__ == 4 && __GNUC_MINOR__ >= 5)
84 #define offsetof(s, m) __builtin_offsetof(s, m)
85 #else
86 #endif /* ! codereview */
87 #if __cplusplus >= 199711L
88 #define offsetof(s, m) (std::size_t)&(((s *)0)->m)
89 #else
90 #define offsetof(s, m) (size_t)&(((s *)0)->m)
91 #endif
92 #endif /* GNUC, etc. */
93 #endif /* ! codereview */

95 #ifdef __cplusplus
96 }
97 #endif

99 #endif /* _ISO_STDDEF_ISO_H */
```

new/usr/src/lib/libumem/common/misc.h

1

```
*****
3536 Thu Feb 25 15:39:36 2016
new/usr/src/lib/libumem/common/misc.h
2976 remove useless offsetof() macros
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */

27 #ifndef _MISC_H
28 #define _MISC_H

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

30 #include <sys/types.h>
31 #include <sys/time.h>
32 #include <thread.h>
33 #include <pthread.h>
34 #include <stdarg.h>
35 #include <stddef.h>
36 #endif /* !codereview */

38 #ifdef __cplusplus
39 extern "C" {
40 #endif

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

47 /*
48  * various utility functions
49  * These are globally implemented.
50  */

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

44 /*
48  * a safe printf -- do not use for error messages.
49  */
50 void debug_printf(const char *format, ...);
```

new/usr/src/lib/libumem/common/misc.h

2

```
52 /*
53  * adds a message to the log without writing it out.
54  */
55 void log_message(const char *format, ...);

57 /*
58  * returns the index of the (high/low) bit + 1
59  */
60 int highbit(ulong_t);
61 int lowbit(ulong_t);
62 #pragma no_side_effect(highbit, lowbit)

64 /*
65  * Converts a hrttime_t to a timestruc_t
66  */
67 void hrt2ts(hrttime_t hrt, timestruc_t *tsp);

69 /*
70  * tries to print out the symbol and offset of a pointer using umem_error_info
71  */
72 int print_sym(void *pointer);

74 /*
75  * Information about the current error. Can be called multiple times, should
76  * be followed eventually with a call to umem_err or umem_err_recoverable.
77  */
78 void umem_printf(const char *format, ...);
79 void umem_vprintf(const char *format, va_list);

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

83 void umem_error_enter(const char *);

85 /*
86  * prints error message and stack trace, then aborts. Cannot return.
87  */
88 void umem_panic(const char *format, ...) __NORETURN;
89 #pragma does_not_return(umem_panic)
90 #pragma rarely_called(umem_panic)

92 /*
93  * like umem_err, but only aborts if umem_abort > 0
94  */
95 void umem_err_recoverable(const char *format, ...);

97 /*
98  * We define our own assertion handling since libc's assert() calls malloc()
99  */
100 #ifdef NDEBUG
101 #define ASSERT(assertion) (void)0
102 #else
103 #define ASSERT(assertion) (void)((assertion) || \
104  __umem_assert_failed(#assertion, __FILE__, __LINE__))
105 #endif

107 int __umem_assert_failed(const char *assertion, const char *file, int line);
108 #pragma does_not_return(__umem_assert_failed)
109 #pragma rarely_called(__umem_assert_failed)
110 /*
111  * These have architecture-specific implementations.
112  */

114 /*
115  * Returns the current function's frame pointer.
116  */
117 extern void *getfp(void);
```

```
119 /*
120 * puts a pc-only stack trace of up to pcstack_limit frames into pcstack.
121 * Returns the number of stacks written.
122 *
123 * if check_sighandler != 0, and we are in a signal context, calls
124 * umem_err_recoverable.
125 */
126 extern int getpcstack(uintptr_t *pcstack, int pcstack_limit,
127     int check_sighandler);
129 #ifdef __cplusplus
130 }
unchanged_portion_omitted
```

```

*****
49098 Thu Feb 25 15:39:36 2016
new/usr/src/lib/lvm/libmeta/common/meta_statconconcise.c
2976 remove useless offsetof() macros
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2006 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

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

26 #include <meta.h>
27 #include <assert.h>
28 #include <ctype.h>
29 #include <mdiox.h>
30 #include <meta.h>
31 #include <stdio.h>
32 #include <stdlib.h>
33 #include <stddef.h>
34 #endif /* ! codereview */
35 #include <strings.h>
36 #include <sys/lvm/md_mddb.h>
37 #include <sys/lvm/md_names.h>
38 #include <sys/lvm/md_crc.h>
39 #include <sys/lvm/md_convert.h>

42 /*
43  * Design Notes:
44  *
45  * All of the code in this file supports the addition of metastat -c output
46  * for the verbose option of metaimport. Some of this code is also used by
47  * the command metastat for concise output(cmd/lvm/util/metastat.c).
48  * The code is designed to produce the same output as metastat -c does for a
49  * given diskset--with a couple exceptions.
50  * The primary differences between the output for the metastat -c command and
51  * metastat output for metaimport -v are:
52  * - the set name is not printed next to each metadvice
53  * - top-level state information is not printed for some metadvicees
54  * - the percent that a disk has completed resyncing is not listed
55  * in metaimport -v.
56  *
57  *
58  * The general layout of this file is as follows:
59  *

```

```

60 * - report_metastat_info()
61 * This is the primary entry point for the functions in this file, with
62 * the exception of several functions that are also called from
63 * cmd/io/lvm/util/metastat.c
64 * report_metastat_info() calls functions to read in all the the
65 * Directory blocks and Record blocks and then process the information
66 * needed to print out the metadvice records in the same format as
67 * metastat -c.
68 *
69 * - read_all_mdrecords()
70 * Reads in all the Directory blocks in the diskset and verifies their
71 * validity. For each Directory block, it loops through all Directory
72 * Entries and for each one that contains a metadvice record calls
73 * read_md_record(). Because the output is designed to imitate the
74 * output of metastat -c, we ignore metadvice records for
75 * optimized resync, changelog, and translog.
76 *
77 * - read_md_record()
78 * Reads in a Directory Entry and its associated Record block. The
79 * revision information for the Record block is checked and it is
80 * determined whether or not it is a 64bit Record block or a 32bit record
81 * block. For each valid Record block, it allocates an md_im_rec_t
82 * structure and calls extract_mduser_data().
83 *
84 * - extract_mduser_data()
85 * Populates the md_im_rec_t data structure with information about the
86 * record's associated metadvice. Also, the name of the metadvice is
87 * either copied from the NM namespace(if it exists there) or is generated
88 * from the record's un_self_id.
89 *
90 * - process_toplevel_devices()
91 * For a given metadvice type, searches through the md_im_rec_t **mdimpp,
92 * list of all metadvicees in the set, to find all records of the
93 * specified type that do not have a parent and puts them on a temp list.
94 * The temp list is then iterated through and the associated processing
95 * function is called.
96 *
97 * - process_(trans, hotspare, hotspare_pool, soft_part, mirror, stripe, raid)
98 * These functions are called by using the dfunc field in the mdimpp list.
99 * Each process function only understands its own type of metadvice. Once
100 * it processes the metadvice it was called for, it then loops through
101 * all of the underlying metadvicees. After printing the name of the
102 * underlying metadvice, it puts in on a list to be processed. If the
103 * underlying device is a physical device, then print_physical_device is
104 * called.
105 * Once all information about the original metadvice is processed, it
106 * loops through the list of underlying metadvicees and calls the
107 * appropriate function to process them.
108 *
109 * - process_toplevel_softparts()
110 * To match the output for metastat -c, all top-level softpartitions
111 * are printed out in groups based on their underlying metadvice--so that
112 * the underlying metadvice only needs to be processed once.
113 *
114 * - meta_get_(sm_state, raid_col_state, stripe_state, hs_state)
115 * These functions are used to retrieve the metadvice state information.
116 * They are also used by the metastat concise routines in
117 * cmd/lvm/util/metastat.c.
118 *
119 */

122 /*
123  * md_im_rec is a doubly linked list used to store the rb_data for each
124  * directory entry that corresponds to a metadvice.
125  * n_key: is set, if there is an associated entry in the NM namespace.

```

```

126 * dfunc: is set to point to the function that processes the particular
127 * metadvice associated with the record.
128 * hs_record_id: is only set, if the metadvice is a hotspare.
129 * un_self_id: is set for all other records. This is also used to generate
130 * the name of the metadvice if there is no entry for the metadvice in
131 * the NM namespace--n_key is not set.
132 */
133 typedef struct md_im_rec {
134     mdkey_t          n_key; /* NM namespace key */
135     struct md_im_rec *next;
136     struct md_im_rec *prev;
137     uint_t          md_type;
138     uint_t          has_parent; /* either 0(no parent) or 1 */
139     minor_t         un_self_id;
140     mddb_recid_t    hs_record_id; /* hotspare recid */
141     char            *n_name; /* name of metadvice */
142     void            (*dfunc) ();
143     ushort_t       record_len;
144     /* pointer to the unit structure for the metadvice, e.g. rb_data[0] */
145     void            *record;
146 } md_im_rec_t;

148 /*
149 * md_im_list is used to group toplevel metadvicees by type and to group
150 * the underlying devices for a particular metadvice.
151 */
152 typedef struct md_im_list {
153     struct md_im_list *next;
154     struct md_im_rec  *mdrec;
155 } md_im_list_t;

158 /*
159 * MAXSIZEMDRECNAME is the value that has historically been used to allocate
160 * space for the metadvice name
161 */
162 #define MAXSIZEMDRECNAME    20
163 #define NAMEWIDTH          16
164 #define offsetof(s, m)    ((size_t)&(((s *)0)->m))
165 #define NOT_PHYSICAL_DEV  0
166 #define PHYSICAL_DEV      1

168 /*
169 * strip_blacks()
170 *
171 * Strip blanks from string.  Used for size field in concise output.
172 */
173 static char *
174 strip_blacks(char *s)
175 {
176     char *p;
177
178     for (p = s; *p; ) {
179         if (*p == ' ') {
180             char *t;
181             for (t = p; *t; t++) {
182                 *t = *(t + 1);
183             }
184         } else {
185             p++;
186         }
187     }
188
189     return (s);
190 }

```

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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 #define _RDC_
27 #include <sys/types.h>
28 #include <sys/ksynch.h>
29 #include <sys/kmem.h>
30 #include <sys/errno.h>
31 #include <sys/conf.h>
32 #include <sys/cmn_err.h>
33 #include <sys/modctl.h>
34 #include <sys/cred.h>
35 #include <sys/ddi.h>
36 #include <sys/sysmacros.h>
37 #endif /* ! codereview */
38 #include <sys/unistat/spcs_s.h>
39 #include <sys/unistat/spcs_s_k.h>
40 #include <sys/unistat/spcs_errors.h>

42 #include <sys/nsc_thread.h>
43 #ifdef DS_DDICT
44 #include "../contract.h"
45 #endif
46 #include <sys/nsctl/nsctl.h>
47 #include <sys/nsctl/nsvers.h>

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

51 #include "rdc.h"
52 #include "rdc_io.h"
53 #include "rdc_bitmap.h"
54 #include "rdc_ioctl.h"
55 #include "rdcsrv.h"
56 #include "rdc_diskq.h"

58 #define DIDINIT          0x01
59 #define DIDNODES        0x02
60 #define DIDCONFIG       0x04

```

```

62 static int rdcopen(dev_t *devp, int flag, int otyp, cred_t *crp);
63 static int rdcclose(dev_t dev, int flag, int otyp, cred_t *crp);
64 static int rdcpri(dev_t dev, char *str);
65 static int rdciocctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *crp,
66         int *rvp);
67 static int rdccattach(dev_info_t *dip, ddi_attach_cmd_t cmd);
68 static int rdcdetach(dev_info_t *dip, ddi_detach_cmd_t cmd);
69 static int rdcgetinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,
70         void **result);
71 #ifdef DEBUG
72 static int rdc_clrkstat(void *);
73 #endif

75 /*
76  * kstat interface
77  */
78 static kstat_t *sndr_kstats;

80 int sndr_info_stats_update(kstat_t *ksp, int rw);

82 static sndr_m_stats_t sndr_info_stats = {
83     {RDC_MKSTAT_MAXSETS,          KSTAT_DATA_ULONG},
84     {RDC_MKSTAT_MAXFBAS,         KSTAT_DATA_ULONG},
85     {RDC_MKSTAT_RPC_TIMEOUT,     KSTAT_DATA_ULONG},
86     {RDC_MKSTAT_HEALTH_THRES,   KSTAT_DATA_ULONG},
87     {RDC_MKSTAT_BITMAP_WRITES,  KSTAT_DATA_ULONG},
88     {RDC_MKSTAT_CLNT_COTS_CALLS, KSTAT_DATA_ULONG},
89     {RDC_MKSTAT_CLNT_CLTS_CALLS, KSTAT_DATA_ULONG},
90     {RDC_MKSTAT_SVC_COTS_CALLS,  KSTAT_DATA_ULONG},
91     {RDC_MKSTAT_SVC_CLTS_CALLS,  KSTAT_DATA_ULONG},
92     {RDC_MKSTAT_BITMAP_REF_DELAY, KSTAT_DATA_ULONG}
93 };

95 int rdc_info_stats_update(kstat_t *ksp, int rw);

97 static rdc_info_stats_t rdc_info_stats = {
98     {RDC_IKSTAT_FLAGS,          KSTAT_DATA_ULONG},
99     {RDC_IKSTAT_SYNCFLAGS,     KSTAT_DATA_ULONG},
100    {RDC_IKSTAT_BMPFLAGS,       KSTAT_DATA_ULONG},
101    {RDC_IKSTAT_SYNCPOS,        KSTAT_DATA_ULONG},
102    {RDC_IKSTAT_VOLSIZE,        KSTAT_DATA_ULONG},
103    {RDC_IKSTAT_BITSSET,        KSTAT_DATA_ULONG},
104    {RDC_IKSTAT_AUTOSYNC,        KSTAT_DATA_ULONG},
105    {RDC_IKSTAT_MAXQFBAS,        KSTAT_DATA_ULONG},
106    {RDC_IKSTAT_MAXQITEMS,      KSTAT_DATA_ULONG},
107    {RDC_IKSTAT_FILE,           KSTAT_DATA_STRING},
108    {RDC_IKSTAT_SECFILE,        KSTAT_DATA_STRING},
109    {RDC_IKSTAT_BITMAP,         KSTAT_DATA_STRING},
110    {RDC_IKSTAT_PRIMARY_HOST,   KSTAT_DATA_STRING},
111    {RDC_IKSTAT_SECONDARY_HOST, KSTAT_DATA_STRING},
112    {RDC_IKSTAT_TYPE_FLAG,      KSTAT_DATA_ULONG},
113    {RDC_IKSTAT_BMP_SIZE,       KSTAT_DATA_ULONG},
114    {RDC_IKSTAT_DISK_STATUS,    KSTAT_DATA_ULONG},
115    {RDC_IKSTAT_IF_DOWN,        KSTAT_DATA_ULONG},
116    {RDC_IKSTAT_IF_RPC_VERSION, KSTAT_DATA_ULONG},
117    {RDC_IKSTAT_ASYNC_BLOCK_HWM, KSTAT_DATA_ULONG},
118    {RDC_IKSTAT_ASYNC_ITEM_HWM, KSTAT_DATA_ULONG},
119    {RDC_IKSTAT_ASYNC_THROTTLE_DELAY, KSTAT_DATA_ULONG},
120    {RDC_IKSTAT_ASYNC_ITEMS,    KSTAT_DATA_ULONG},
121    {RDC_IKSTAT_ASYNC_BLOCKS,   KSTAT_DATA_ULONG},
122    {RDC_IKSTAT_QUEUE_TYPE,     KSTAT_DATA_CHAR}
123 };

125 static struct cb_ops rdc_cb_ops = {
126     rdcopen,
127     rdcclose,

```

```

128     nulldev,          /* no strategy */
129     rdcprint,
130     nodev,           /* no dump */
131     nodev,           /* no read */
132     nodev,           /* no write */
133     rdciocctl,
134     nodev,           /* no devmap */
135     nodev,           /* no mmap */
136     nodev,           /* no segmap */
137     nochpoll,
138     ddi_prop_op,
139     NULL,            /* not STREAMS */
140     D_NEW | D_MP | D_64BIT,
141     CB_REV,
142     nodev,           /* no aread */
143     nodev,           /* no awrite */
144 };

146 static struct dev_ops rdc_ops = {
147     DEVO_REV,
148     0,
149     rdcgetinfo,
150     nulldev,         /* identify */
151     nulldev,         /* probe */
152     rdcattach,
153     rdcdetach,
154     nodev,           /* no reset */
155     &rdc_cb_ops,
156     (struct bus_ops *)NULL
157 };

159 static struct modldrv rdc_ldrv = {
160     &mod_driverops,
161     "nws:Remote Mirror:" ISS_VERSION_STR,
162     &rdc_ops
163 };

165 static struct modlinkage rdc_modlinkage = {
166     MODREV_1,
167     &rdc_ldrv,
168     NULL
169 };

171 const int sndr_major_rev = ISS_VERSION_MAJ;
172 const int sndr_minor_rev = ISS_VERSION_MIN;
173 const int sndr_micro_rev = ISS_VERSION_MIC;
174 const int sndr_baseline_rev = ISS_VERSION_NUM;
175 static char sndr_version[16];

177 static void *rdc_dip;

179 extern int _rdc_init_dev();
180 extern void _rdc_deinit_dev();
181 extern void rdc_link_down_free();

183 int rdc_bitmap_mode;
184 int rdc_auto_sync;
185 int rdc_max_sets;
186 extern int rdc_health_thres;

188 kmutex_t rdc_sync_mutex;
189 rdc_sync_event_t rdc_sync_event;
190 clock_t rdc_sync_event_timeout;

192 static void
193 rdc_sync_event_init()

```

```

194 {
195     mutex_init(&rdc_sync_mutex, NULL, MUTEX_DRIVER, NULL);
196     mutex_init(&rdc_sync_event.mutex, NULL, MUTEX_DRIVER, NULL);
197     cv_init(&rdc_sync_event.cv, NULL, CV_DRIVER, NULL);
198     cv_init(&rdc_sync_event.done_cv, NULL, CV_DRIVER, NULL);
199     rdc_sync_event.master[0] = 0;
200     rdc_sync_event.lbolt = (clock_t)0;
201     rdc_sync_event.timeout = RDC_SYNC_EVENT_TIMEOUT;
202 }

205 static void
206 rdc_sync_event_destroy()
207 {
208     mutex_destroy(&rdc_sync_mutex);
209     mutex_destroy(&rdc_sync_event.mutex);
210     cv_destroy(&rdc_sync_event.cv);
211     cv_destroy(&rdc_sync_event.done_cv);
212 }

216 int
217 _init(void)
218 {
219     return (mod_install(&rdc_modlinkage));
220 }

222 int
223 _fini(void)
224 {
225     return (mod_remove(&rdc_modlinkage));
226 }

228 int
229 _info(struct modinfo *modinfop)
230 {
231     return (mod_info(&rdc_modlinkage, modinfop));
232 }

234 static int
235 rdcattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
236 {
237     intptr_t flags;
238     int instance;
239     int i;

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

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

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

249     instance = ddi_get_instance(dip);
250     rdc_dip = dip;

252     flags = 0;

254     rdc_sync_event_init();

256     /*
257      * rdc_max_sets must be set before calling _rdc_load().
258      */

```



```

260 rdc_max_sets = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
261 DDI_PROP_DONTPASS | DDI_PROP_NOTPROM, "rdc_max_sets", 64);
263
264 if (_rdc_init_dev()) {
265     cmn_err(CE_WARN, "!rdc: _rdc_init_dev failed");
266     goto out;
267 }
268 flags |= DIDINIT;
269
270 if (_rdc_load() != 0) {
271     cmn_err(CE_WARN, "!rdc: _rdc_load failed");
272     goto out;
273 }
274
275 if (_rdc_configure()) {
276     cmn_err(CE_WARN, "!rdc: _rdc_configure failed");
277     goto out;
278 }
279 flags |= DIDCONFIG;
280
281 if (ddi_create_minor_node(dip, "rdc", S_IFCHR, instance, DDI_PSEUDO, 0)
282 != DDI_SUCCESS) {
283     cmn_err(CE_WARN, "!rdc: could not create node.");
284     goto out;
285 }
286 flags |= DIDNODES;
287
288 rdc_bitmap_mode = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
289 DDI_PROP_DONTPASS | DDI_PROP_NOTPROM,
290 "rdc_bitmap_mode", 0);
291
292 switch (rdc_bitmap_mode) {
293 case RDC_BMP_AUTO:           /* 0 */
294     break;
295 case RDC_BMP_ALWAYS:       /* 1 */
296     break;
297 case RDC_BMP_NEVER:        /* 2 */
298     cmn_err(CE_NOTE, "!SNDR bitmap mode override");
299     cmn_err(CE_CONT,
300 "!SNDR: bitmaps will only be written on shutdown\n");
301     break;
302 default:                    /* unknown */
303     cmn_err(CE_NOTE,
304 "!SNDR: unknown bitmap mode %d - autodetecting mode",
305 rdc_bitmap_mode);
306     rdc_bitmap_mode = RDC_BMP_AUTO;
307     break;
308 }
309
310 rdc_bitmap_init();
311
312 rdc_auto_sync = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
313 DDI_PROP_DONTPASS | DDI_PROP_NOTPROM,
314 "rdc_auto_sync", 0);
315
316 i = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
317 DDI_PROP_DONTPASS | DDI_PROP_NOTPROM,
318 "rdc_health_thres", RDC_HEALTH_THRESHOLD);
319 if (i >= RDC_MIN_HEALTH_THRES)
320     rdc_health_thres = i;
321 else
322     cmn_err(CE_WARN, "!value rdc_heath_thres from rdc.conf ignored "
323 "as it is smaller than the min value of %d",
324 RDC_MIN_HEALTH_THRES);
325
326 ddi_set_driver_private(dip, (caddr_t)flags);

```

```

326 ddi_report_dev(dip);
327
328 sndr_kstats = kstat_create(RDC_KSTAT_MODULE, 0,
329 RDC_KSTAT_MINFO, RDC_KSTAT_CLASS, KSTAT_TYPE_NAMED,
330 sizeof(sndr_m_stats_t) / sizeof(kstat_named_t),
331 KSTAT_FLAG_VIRTUAL);
332
333 if (sndr_kstats) {
334     sndr_kstats->ks_data = &sndr_info_stats;
335     sndr_kstats->ks_update = sndr_info_stats.update;
336     sndr_kstats->ks_private = &rdc_k_info[0];
337     kstat_install(sndr_kstats);
338 } else
339     cmn_err(CE_WARN, "!SNDR: module kstats failed");
340
341 return (DDI_SUCCESS);
342
343 out:
344 DTRACE_PROBE(rdc_attach_failed);
345 ddi_set_driver_private(dip, (caddr_t)flags);
346 (void) rdcdetach(dip, DDI_DETACH);
347 return (DDI_FAILURE);
348 }
349
350 static int
351 rdcdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
352 {
353     rdc_k_info_t *krdc;
354     rdc_u_info_t *urdc;
355     int rdcd;
356     intptr_t flags;
357
358     if (cmd != DDI_DETACH) {
359         DTRACE_PROBE(rdc_detach_unknown_cmd);
360         return (DDI_FAILURE);
361     }
362
363     if (rdc_k_info == NULL || rdc_u_info == NULL)
364         goto cleanup;
365
366     mutex_enter(&rdc_conf_lock);
367
368     for (rdcd = 0; rdcd < rdc_max_sets; rdcd++) {
369         krdc = &rdc_k_info[rdcd];
370         urdc = &rdc_u_info[rdcd];
371
372         if (IS_ENABLED(urdc) || krdc->devices) {
373             #ifndef DEBUG
374                 cmn_err(CE_WARN,
375 "!rdc: cannot detach, rdc %d still in use", rdcd);
376             #endif
377             mutex_exit(&rdc_conf_lock);
378             DTRACE_PROBE(rdc_detach_err_busy);
379             return (DDI_FAILURE);
380         }
381     }
382
383     mutex_exit(&rdc_conf_lock);
384
385 cleanup:
386     flags = (intptr_t)ddi_get_driver_private(dip);
387
388     if (flags & DIDNODES)
389         ddi_remove_minor_node(dip, NULL);

```

```

392     if (sndr_kstats) {
393         kstat_delete(sndr_kstats);
394     }
395     if (flags & DIDINIT)
396         _rdc_deinit_dev();
397
398     if (flags & DIDCONFIG) {
399         (void) _rdc_deconfigure();
400         (void) _rdc_unload();
401         rdcsrv_unload();
402     }
403
404     rdc_sync_event_destroy();
405     rdc_link_down_free();
406
407     rdc_dip = NULL;
408     return (DDI_SUCCESS);
409 }
410
411 /* ARGSUSED */
412 static int
413 rdcgetinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
414 {
415     int rc = DDI_FAILURE;
416
417     switch (infocmd) {
418
419     case DDI_INFO_DEVT2DEVINFO:
420         *result = rdc_dip;
421         rc = DDI_SUCCESS;
422         break;
423
424     case DDI_INFO_DEVT2INSTANCE:
425         /* We only have a single instance */
426         *result = 0;
427         rc = DDI_SUCCESS;
428         break;
429
430     default:
431         break;
432     }
433
434     return (rc);
435 }
436
437 /* ARGSUSED */
438
439 static int
440 rdcclose(dev_t devp, int flag, int otyp, cred_t *crp)
441 {
442     return (0);
443 }
444
445 /* ARGSUSED */
446
447 static int
448 rdcclose(dev_t dev, int flag, int otyp, cred_t *crp)
449 {
450     return (0);
451 }
452
453 /* ARGSUSED */
454
455 static int

```

```

458 rdcprint(dev_t dev, char *str)
459 {
460     int instance = 0;
461
462     cmn_err(CE_WARN, "!rdc%d: %s", instance, str);
463     return (0);
464 }
465
466 static int
467 convert_ioctl_args(int cmd, intptr_t arg, int mode, _rdc_ioctl_t *args)
468 {
469     _rdc_ioctl32_t args32;
470
471     if (ddi_copyin((void *)arg, &args32, sizeof (_rdc_ioctl32_t), mode))
472         return (EFAULT);
473
474     bzero((void *)args, sizeof (_rdc_ioctl_t));
475
476     switch (cmd) {
477     case RDC_CONFIG:
478         args->arg0 = (uint32_t)args32.arg0; /* _rdc_config_t * */
479         args->arg1 = (uint32_t)args32.arg1; /* pointer */
480         args->arg2 = (uint32_t)args32.arg2; /* size */
481         args->ustatus = (spsc_s_info_t)args32.ustatus;
482         break;
483
484     case RDC_STATUS:
485         args->arg0 = (uint32_t)args32.arg0; /* pointer */
486         args->ustatus = (spsc_s_info_t)args32.ustatus;
487         break;
488
489     case RDC_ENABLE_SVR:
490         args->arg0 = (uint32_t)args32.arg0; /* _rdc_svc_args * */
491         break;
492
493     case RDC_VERSION:
494         args->arg0 = (uint32_t)args32.arg0; /* _rdc_version_t * */
495         args->ustatus = (spsc_s_info_t)args32.ustatus;
496         break;
497
498     case RDC_SYNC_EVENT:
499         args->arg0 = (uint32_t)args32.arg0; /* char * */
500         args->arg1 = (uint32_t)args32.arg1; /* char * */
501         args->ustatus = (spsc_s_info_t)args32.ustatus;
502         break;
503
504     case RDC_LINK_DOWN:
505         args->arg0 = (uint32_t)args32.arg0; /* char * */
506         args->ustatus = (spsc_s_info_t)args32.ustatus;
507         break;
508
509     case RDC_POOL_CREATE:
510         args->arg0 = (uint32_t)args32.arg0; /* svcpool_args * */
511         break;
512
513     case RDC_POOL_WAIT:
514         args->arg0 = (uint32_t)args32.arg0; /* int */
515         break;
516
517     case RDC_POOL_RUN:
518         args->arg0 = (uint32_t)args32.arg0; /* int */
519         break;
520
521     default:
522         return (EINVAL);
523     }
524
525     return (0);

```

```
524 }

37 /*
38  * Yet another standard thing that is not standard ...
39  */
40 #ifndef offsetof
41 #define offsetof(s, m) ((size_t)&((s *)0)->m)
42 #endif

526 /*
527  * Build a 32bit rdc_set structure and copyout to the user level.
528  */
529 int
530 rdc_status_copy32(const void *arg, void *usetp, size_t size, int mode)
531 {
532     rdc_u_info_t *urdc = (rdc_u_info_t *)arg;
533     struct rdc_set32 set32;
534     size_t tailsize;
535 #ifdef DEBUG
536     size_t tailsize32;
537 #endif

539     bzero(&set32, sizeof (set32));

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

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

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

550     /*
551     * the rest, avoiding netconfig
552     * note: the tail must be the same size in both structures
553     */
554     tailsize = sizeof (struct rdc_set) - offsetof(struct rdc_set, flags);
555 #ifdef DEBUG
556     /*
557     * ASSERT is calling for debug reason, and tailsize32 is only declared
558     * for ASSERT, put them under debug to avoid lint warning.
559     */
560     tailsize32 = sizeof (struct rdc_set32) -
561         offsetof(struct rdc_set32, flags);
562     ASSERT(tailsize == tailsize32);
563 #endif

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

567     /* copyout to user level */
568     return (ddi_copyout(&set32, usetp, size, mode));
569 }
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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 #include <sys/types.h>
27 #include <sys/ksynch.h>
28 #include <sys/cm_n_err.h>
29 #include <sys/kmem.h>
30 #include <sys/conf.h>
31 #include <sys/errno.h>
32 #include <sys/sysmacros.h>
33 #endif /* ! codereview */

35 #ifdef _SunOS_5_6
36 /*
37  * on 2.6 both dki_lock.h and rpc/types.h define bool_t so we
38  * define enum_t here as it is all we need from rpc/types.h
39  * anyway and make it look like we included it. Yuck.
40  */
41 #define _RPC_TYPES_H
42 typedef int enum_t;
43 #else
44 #ifndef DS_DDICT
45 #include <rpc/types.h>
46 #endif
47 #endif /* _SunOS_5_6 */

49 #include <sys/ddi.h>

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

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

56 #include "rdc_io.h"
57 #include "rdc_bitmap.h"
58 #include "rdc_update.h"
59 #include "rdc_ioctl.h"
60 #include "rdcsrv.h"
61 #include "rdc_diskq.h"

```

```

63 #include <sys/unistat/spcs_s.h>
64 #include <sys/unistat/spcs_s_k.h>
65 #include <sys/unistat/spcs_errors.h>

67 volatile int net_exit;
68 nsc_size_t MAX_RDC_FBAS;

70 #ifdef DEBUG
71 int RDC_MAX_SYNC_THREADS = 8;
72 int rdc_maxthreads_last = 8;
73 #endif

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

78 /*
79  * rdc_conf_lock is used as a global device configuration lock.
80  * It is also used by enable/resume and disable/suspend code to ensure that
81  * the transition of an rdc set between configured and unconfigured is
82  * atomic.
83  *
84  * krdc->group->lock is used to protect state changes of a configured rdc
85  * set (e.g. changes to urdc->flags), such as enabled to disabled and vice
86  * versa.
87  *
88  * rdc_many_lock is also used to protect changes in group membership. A group
89  * linked list cannot change while this lock is held. The many list and the
90  * multi-hop list are both protected by rdc_many_lock.
91  */
92 kmutex_t rdc_conf_lock;
93 kmutex_t rdc_many_lock;          /* Many/multi-list lock */

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

99 static int rdc_net_hnd_id = 1;

101 extern kmutex_t rdc_clnt_lock;

103 static void rdc_ditemfree(rdc_net_dataset_t *);
104 void rdc_clnt_destroy(void);

106 rdc_k_info_t *rdc_k_info;
107 rdc_u_info_t *rdc_u_info;

109 unsigned long rdc_async_timeout;

111 nsc_size_t rdc_maxthres_queue = RDC_MAXTHRES_QUEUE;
112 int rdc_max_qitems = RDC_MAX_QITEMS;
113 int rdc_asyncthr = RDC_ASYNCTHR;
114 static nsc_svc_t *rdc_volume_update;
115 static int rdc_prealloc_handle = 1;

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

120 /*
121  * Forward declare all statics that are used before defined
122  * to enforce parameter checking
123  *
124  * Some (if not all) of these could be removed if the code were reordered
125  */

127 static void rdc_volume_update_svc(intptr_t);

```

```

128 static void halt_sync(rdc_k_info_t *krdc);
129 void rdc_kstat_create(int index);
130 void rdc_kstat_delete(int index);
131 static int rdc_checkforbitmap(int, nsc_off_t);
132 static int rdc_installbitmap(int, void *, int, nsc_off_t, int, int *, int);
133 static rdc_group_t *rdc_newgroup();

135 int rdc_enable_diskq(rdc_k_info_t *krdc);
136 void rdc_close_diskq(rdc_group_t *group);
137 int rdc_suspend_diskq(rdc_k_info_t *krdc);
138 int rdc_resume_diskq(rdc_k_info_t *krdc);
139 void rdc_init_diskq_header(rdc_group_t *grp, dqheader *header);
140 void rdc_fail_diskq(rdc_k_info_t *krdc, int wait, int dolog);
141 void rdc_unfail_diskq(rdc_k_info_t *krdc);
142 void rdc_unintercept_diskq(rdc_group_t *grp);
143 int rdc_stamp_diskq(rdc_k_info_t *krdc, int rsvrd, int flags);
144 void rdc_qfiller_thr(rdc_k_info_t *krdc);

146 nstset_t *_rdc_ioset;
147 nstset_t *_rdc_flset;

149 /*
150  * RDC threadset tunables
151  */
152 int rdc_threads = 64;          /* default number of threads */
153 int rdc_threads_inc = 8;      /* increment for changing the size of the set */

155 /*
156  * Private threadset manipulation variables
157  */
158 static int rdc_threads_hysteresis = 2;
159 /* hysteresis for threadset resizing */
160 static int rdc_sets_active;    /* number of sets currently enabled */

162 #ifdef DEBUG
163 kmutex_t rdc_cntlock;
164 #endif

166 /*
167  * rdc_thread_deconfigure - rdc is being deconfigured, stop any
168  * thread activity.
169  * Inherently single-threaded by the Solaris module unloading code.
170  */
171 static void
172 rdc_thread_deconfigure(void)
173 {
174     nst_destroy(_rdc_ioset);
175     _rdc_ioset = NULL;

178     nst_destroy(_rdc_flset);
179     _rdc_flset = NULL;

181     nst_destroy(sync_info.rdc_syncset);
182     sync_info.rdc_syncset = NULL;
183 }

185 /*
186  * rdc_thread_configure - rdc is being configured, initialize the
187  * threads we need for flushing aync volumes.
188  * Must be called with rdc_conf_lock held.
189  */
190 static int
191 rdc_thread_configure(void)
192 {

```

```

194     ASSERT(MUTEX_HELD(&rdc_conf_lock));

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

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

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

206     return (0);
207 }

210 /*
211  * rdc_thread_tune - called to tune the size of the rdc threadset.
212  * Called from the config code when an rdc_set has been enabled or disabled.
213  * 'sets' is the increment to the number of active rdc_sets.
214  * Must be called with rdc_conf_lock held.
215  */
216 static void
217 rdc_thread_tune(int sets)
218 {
219     int incr = (sets > 0) ? 1 : -1;
220     int change = 0;
221     int nthreads;

225     ASSERT(MUTEX_HELD(&rdc_conf_lock));

227     if (sets < 0)
228         sets = -sets;

230     while (sets-- > 0) {
231         nthreads = nst_nthread(_rdc_ioset);
232         rdc_sets_active += incr;

234         if (rdc_sets_active >= nthreads)
235             change += nst_add_thread(_rdc_ioset, rdc_threads_inc);
236         else if ((rdc_sets_active <
237             (nthreads - (rdc_threads_inc + rdc_threads_hysteresis))) &&
238             ((nthreads - rdc_threads_inc) >= rdc_threads))
239             change -= nst_del_thread(_rdc_ioset, rdc_threads_inc);
240     }

242 #ifdef DEBUG
243     if (change) {
244         cmn_err(CE_NOTE, "!rdc_thread_tune: "
245             "nsets %d, nthreads %d, nthreads change %d",
246             rdc_sets_active, nst_nthread(_rdc_ioset), change);
247     }
248 #endif
249 }

252 /*
253  * _rdc_unload() - cache is being unloaded,
254  * deallocate any dual copy structures allocated during cache
255  * loading.
256  */
257 void
258 _rdc_unload(void)
259 {

```

```

260     int i;
261     rdc_k_info_t *krdc;

263     if (rdc_volume_update) {
264         (void) nsc_unregister_svc(rdc_volume_update);
265         rdc_volume_update = NULL;
266     }

268     rdc_thread_deconfigure();

270     if (rdc_k_info != NULL) {
271         for (i = 0; i < rdc_max_sets; i++) {
272             krdc = &rdc_k_info[i];
273             mutex_destroy(&krdc->dc_sleep);
274             mutex_destroy(&krdc->bmapmutex);
275             mutex_destroy(&krdc->kstat_mutex);
276             mutex_destroy(&krdc->bmp_kstat_mutex);
277             mutex_destroy(&krdc->syncbitmutex);
278             cv_destroy(&krdc->busycv);
279             cv_destroy(&krdc->closingcv);
280             cv_destroy(&krdc->haltcv);
281             cv_destroy(&krdc->synccv);
282         }
283     }

285     mutex_destroy(&sync_info.lock);
286     mutex_destroy(&rdc_ping_lock);
287     mutex_destroy(&net_blk_lock);
288     mutex_destroy(&rdc_conf_lock);
289     mutex_destroy(&rdc_many_lock);
290     mutex_destroy(&rdc_net_hnd_id_lock);
291     mutex_destroy(&rdc_clnt_lock);
292 #ifdef DEBUG
293     mutex_destroy(&rdc_cntlock);
294 #endif
295     net_exit = ATM_EXIT;

297     if (rdc_k_info != NULL)
298         kmem_free(rdc_k_info, sizeof (*rdc_k_info) * rdc_max_sets);
299     if (rdc_u_info != NULL)
300         kmem_free(rdc_u_info, sizeof (*rdc_u_info) * rdc_max_sets);
301     rdc_k_info = NULL;
302     rdc_u_info = NULL;
303     rdc_max_sets = 0;
304 }

307 /*
308  * _rdc_load() - rdc is being loaded, Allocate anything
309  * that will be needed while the cache is loaded but doesn't really
310  * depend on configuration parameters.
311  */
312 */
313 int
314 _rdc_load(void)
315 {
316     int i;
317     rdc_k_info_t *krdc;

319     mutex_init(&rdc_ping_lock, NULL, MUTEX_DRIVER, NULL);
320     mutex_init(&net_blk_lock, NULL, MUTEX_DRIVER, NULL);
321     mutex_init(&rdc_conf_lock, NULL, MUTEX_DRIVER, NULL);
322     mutex_init(&rdc_many_lock, NULL, MUTEX_DRIVER, NULL);
323     mutex_init(&rdc_net_hnd_id_lock, NULL, MUTEX_DRIVER, NULL);
324     mutex_init(&rdc_clnt_lock, NULL, MUTEX_DRIVER, NULL);
325     mutex_init(&sync_info.lock, NULL, MUTEX_DRIVER, NULL);

```

```

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

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

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

341     rdc_u_info = kmem_zalloc(sizeof (*rdc_u_info) * rdc_max_sets, KM_SLEEP);
342     if (!rdc_u_info) {
343         kmem_free(rdc_k_info, sizeof (*rdc_k_info) * rdc_max_sets);
344         return (ENOMEM);
345     }

347     net_exit = ATM_NONE;
348     for (i = 0; i < rdc_max_sets; i++) {
349         krdc = &rdc_k_info[i];
350         bzero(krdc, sizeof (*krdc));
351         krdc->index = i;
352         mutex_init(&krdc->dc_sleep, NULL, MUTEX_DRIVER, NULL);
353         mutex_init(&krdc->bmapmutex, NULL, MUTEX_DRIVER, NULL);
354         mutex_init(&krdc->kstat_mutex, NULL, MUTEX_DRIVER, NULL);
355         mutex_init(&krdc->bmp_kstat_mutex, NULL, MUTEX_DRIVER, NULL);
356         mutex_init(&krdc->syncbitmutex, NULL, MUTEX_DRIVER, NULL);
357         cv_init(&krdc->busycv, NULL, CV_DRIVER, NULL);
358         cv_init(&krdc->closingcv, NULL, CV_DRIVER, NULL);
359         cv_init(&krdc->haltcv, NULL, CV_DRIVER, NULL);
360         cv_init(&krdc->synccv, NULL, CV_DRIVER, NULL);
361     }

363     rdc_volume_update = nsc_register_svc("RDCVolumeUpdated",
364         rdc_volume_update_svc);

366     return (0);
367 }

369 static void
370 rdc_u_init(rdc_u_info_t *urdc)
371 {
372     const int index = (int)(urdc - &rdc_u_info[0]);

374     if (urdc->secondary.addr.maxlen)
375         free_rdc_netbuf(&urdc->secondary.addr);
376     if (urdc->primary.addr.maxlen)
377         free_rdc_netbuf(&urdc->primary.addr);

379     bzero(urdc, sizeof (rdc_u_info_t));

381     urdc->index = index;
382     urdc->maxqfbas = rdc_maxthres_queue;
383     urdc->maxqitems = rdc_max_qitems;
384     urdc->asynchthr = rdc_asynchthr;
385 }

387 /*
388  * _rdc_configure() - cache is being configured.
389  * Initialize dual copy structures
390  */

```

```

392 int
393 _rdc_configure(void)
394 {
395     int index;
396     rdc_k_info_t *krdc;
397
398     for (index = 0; index < rdc_max_sets; index++) {
399         krdc = &rdc_k_info[index];
400
401         krdc->remote_index = -1;
402         krdc->dcio_bitmap = NULL;
403         krdc->bitmap_ref = NULL;
404         krdc->bitmap_size = 0;
405         krdc->bitmap_write = 0;
406         krdc->disk_status = 0;
407         krdc->many_next = krdc;
408
409         rdc_u_init(&rdc_u_info[index]);
410     }
411
412     rdc_async_timeout = 120 * HZ; /* Seconds * HZ */
413     MAX_RDC_FBAS = FBA_LEN(RDC_MAXDATA);
414     if (net_exit != ATM_INIT) {
415         net_exit = ATM_INIT;
416         return (0);
417     }
418     return (0);
419 }
420
421 /*
422 * _rdc_deconfigure - rdc is being deconfigured, shut down any
423 * dual copy operations and return to an unconfigured state.
424 */
425 void
426 _rdc_deconfigure(void)
427 {
428     rdc_k_info_t *krdc;
429     rdc_u_info_t *urdc;
430     int index;
431
432     for (index = 0; index < rdc_max_sets; index++) {
433         krdc = &rdc_k_info[index];
434         urdc = &rdc_u_info[index];
435
436         krdc->remote_index = -1;
437         krdc->dcio_bitmap = NULL;
438         krdc->bitmap_ref = NULL;
439         krdc->bitmap_size = 0;
440         krdc->bitmap_write = 0;
441         krdc->disk_status = 0;
442         krdc->many_next = krdc;
443
444         if (urdc->primary.addr.maxlen)
445             free_rdc_netbuf(&(urdc->primary.addr));
446
447         if (urdc->secondary.addr.maxlen)
448             free_rdc_netbuf(&(urdc->secondary.addr));
449
450         bzero(urdc, sizeof (rdc_u_info_t));
451         urdc->index = index;
452     }
453     net_exit = ATM_EXIT;
454     rdc_clnt_destroy();
455 }

```

```

459 /*
460 * Lock primitives, containing checks that lock ordering isn't broken
461 */
462 /*ARGSUSED*/
463 void
464 rdc_many_enter(rdc_k_info_t *krdc)
465 {
466     ASSERT(!MUTEX_HELD(&krdc->bmapmutex));
467
468     mutex_enter(&rdc_many_lock);
469 }
470
471 /* ARGSUSED */
472 void
473 rdc_many_exit(rdc_k_info_t *krdc)
474 {
475     mutex_exit(&rdc_many_lock);
476 }
477
478 void
479 rdc_group_enter(rdc_k_info_t *krdc)
480 {
481     ASSERT(!MUTEX_HELD(&rdc_many_lock));
482     ASSERT(!MUTEX_HELD(&rdc_conf_lock));
483     ASSERT(!MUTEX_HELD(&krdc->bmapmutex));
484
485     mutex_enter(&krdc->group->lock);
486 }
487
488 void
489 rdc_group_exit(rdc_k_info_t *krdc)
490 {
491     mutex_exit(&krdc->group->lock);
492 }
493
494 /*
495 * Suspend and disable operations use this function to wait until it is safe
496 * to do continue, without trashing data structures used by other ioctls.
497 */
498 static void
499 wait_busy(rdc_k_info_t *krdc)
500 {
501     ASSERT(MUTEX_HELD(&rdc_conf_lock));
502
503     while (krdc->busy_count > 0)
504         cv_wait(&krdc->busycv, &rdc_conf_lock);
505 }
506
507 /*
508 * Other ioctls use this function to hold off disable and suspend.
509 */
510 void
511 set_busy(rdc_k_info_t *krdc)
512 {
513     ASSERT(MUTEX_HELD(&rdc_conf_lock));
514
515     wait_busy(krdc);
516
517     krdc->busy_count++;
518 }
519
520 /*
521 * Other ioctls use this function to allow disable and suspend to continue.

```

```

524 */
525 void
526 wakeup_busy(rdc_k_info_t *krdc)
527 {
528     ASSERT(MUTEX_HELD(&rdc_conf_lock));
529
530     if (krdc->busy_count <= 0)
531         return;
532
533     krdc->busy_count--;
534     cv_broadcast(&krdc->busycv);
535 }
536
537
538 /*
539  * Remove the rdc set from its group, and destroy the group if no longer in
540  * use.
541  */
542 static void
543 remove_from_group(rdc_k_info_t *krdc)
544 {
545     rdc_k_info_t *p;
546     rdc_group_t *group;
547
548     ASSERT(MUTEX_HELD(&rdc_conf_lock));
549
550     rdc_many_enter(krdc);
551     group = krdc->group;
552
553     group->count--;
554
555     /*
556      * lock queue while looking at thrnum
557      */
558     mutex_enter(&group->ra_queue.net_qlock);
559     if ((group->rdc_thrnum == 0) && (group->count == 0)) {
560
561         /*
562          * Assure the we've stopped and the flusher thread has not
563          * fallen back to sleep
564          */
565         if (krdc->group->ra_queue.qfill_sleeping != RDC_QFILL_DEAD) {
566             group->ra_queue.qfflags |= RDC_QFILLSTOP;
567             while (krdc->group->ra_queue.qfflags & RDC_QFILLSTOP) {
568                 if (krdc->group->ra_queue.qfill_sleeping ==
569                     RDC_QFILL_ASLEEP)
570                     cv_broadcast(&group->ra_queue.qfcv);
571                 mutex_exit(&group->ra_queue.net_qlock);
572                 delay(2);
573                 mutex_enter(&group->ra_queue.net_qlock);
574             }
575         }
576         mutex_exit(&group->ra_queue.net_qlock);
577
578         mutex_enter(&group->diskqmutex);
579         rdc_close_diskq(group);
580         mutex_exit(&group->diskqmutex);
581         rdc_delgroup(group);
582         rdc_many_exit(krdc);
583         krdc->group = NULL;
584         return;
585     }
586     mutex_exit(&group->ra_queue.net_qlock);
587     /*
588      * Always clear the group field.
589      * no, you need it set in rdc_flush_memq().

```

```

590     * to call rdc_group_log()
591     * krdc->group = NULL;
592     */
593
594     /* Take this rdc structure off the group list */
595
596     for (p = krdc->group_next; p->group_next != krdc; p = p->group_next)
597         ;
598     p->group_next = krdc->group_next;
599
600     rdc_many_exit(krdc);
601 }
602
603
604 /*
605  * Add the rdc set to its group, setting up a new group if it's the first one.
606  */
607 static int
608 add_to_group(rdc_k_info_t *krdc, int options, int cmd)
609 {
610     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
611     rdc_u_info_t *utmp;
612     rdc_k_info_t *ktmp;
613     int index;
614     rdc_group_t *group;
615     int rc = 0;
616     nsthread_t *trc;
617
618     ASSERT(MUTEX_HELD(&rdc_conf_lock));
619
620     /*
621      * Look for matching group name, primary host name and secondary
622      * host name.
623      */
624
625     rdc_many_enter(krdc);
626     for (index = 0; index < rdc_max_sets; index++) {
627         utmp = &rdc_u_info[index];
628         ktmp = &rdc_k_info[index];
629
630         if (urdc->group_name[0] == 0)
631             break;
632
633         if (!IS_CONFIGURED(ktmp))
634             continue;
635
636         if (strcmp(utmp->group_name, urdc->group_name,
637                 NSC_MAXPATH) != 0)
638             continue;
639         if (strcmp(utmp->primary_intf, urdc->primary_intf,
640                 MAX_RDC_HOST_SIZE) != 0) {
641             /* Same group name, different primary interface */
642             rdc_many_exit(krdc);
643             return (-1);
644         }
645         if (strcmp(utmp->secondary_intf, urdc->secondary_intf,
646                 MAX_RDC_HOST_SIZE) != 0) {
647             /* Same group name, different secondary interface */
648             rdc_many_exit(krdc);
649             return (-1);
650         }
651     }
652
653     /* Group already exists, so add this set to the group */
654
655     if (((options & RDC_OPT_ASYNC) == 0) &&
        ((ktmp->type_flag & RDC_ASYNCMODE) != 0)) {

```



```

656         /* Must be same mode as existing group members */
657         rdc_many_exit(krdc);
658         return (-1);
659     }
660     if (((options & RDC_OPT_ASYNC) != 0) &&
661         ((ktmp->type_flag & RDC_ASYNCMODE) == 0)) {
662         /* Must be same mode as existing group members */
663         rdc_many_exit(krdc);
664         return (-1);
665     }
666
667     /* cannot reconfigure existing group into new queue this way */
668     if ((cmd != RDC_CMD_RESUME) &&
669         !RDC_IS_DISKQ(ktmp->group) && urdc->disk_queue[0] != '\0') {
670         rdc_many_exit(krdc);
671         return (RDC_EQNOADD);
672     }
673
674     ktmp->group->count++;
675     krdc->group = ktmp->group;
676     krdc->group_next = ktmp->group_next;
677     ktmp->group_next = krdc;
678
679     urdc->autosync = utmp->autosync;        /* Same as rest */
680
681     (void) strncpy(urdc->disk_queue, utmp->disk_queue, NSC_MAXPATH);
682
683     rdc_many_exit(krdc);
684     return (0);
685 }
686
687 /* This must be a new group */
688 group = rdc_newgroup();
689 krdc->group = group;
690 krdc->group_next = krdc;
691 urdc->autosync = -1;    /* Unknown */
692
693 /*
694  * Tune the thread set by one for each thread created
695  */
696 rdc_thread_tune(1);
697
698 trc = nst_create(_rdc_ioset, rdc_qfiller_thr, (void *)krdc, NST_SLEEP);
699 if (trc == NULL) {
700     rc = -1;
701     cmn_err(CE_NOTE, "!unable to create queue filler daemon");
702     goto fail;
703 }
704
705 if (urdc->disk_queue[0] == '\0') {
706     krdc->group->flags |= RDC_MEMQUE;
707 } else {
708     krdc->group->flags |= RDC_DISKQUE;
709
710     /* XXX check here for resume or enable and act accordingly */
711
712     if (cmd == RDC_CMD_RESUME) {
713         rc = rdc_resume_diskq(krdc);
714
715     } else if (cmd == RDC_CMD_ENABLE) {
716         rc = rdc_enable_diskq(krdc);
717         if ((rc == RDC_EQNOADD) && (cmd != RDC_CMD_ENABLE)) {
718             cmn_err(CE_WARN, "!disk queue %s enable failed,"
719                 " enabling memory queue",
720                 urdc->disk_queue);
721             krdc->group->flags &= ~RDC_DISKQUE;

```

```

722         krdc->group->flags |= RDC_MEMQUE;
723         bzero(urdc->disk_queue, NSC_MAXPATH);
724     }
725 }
726 }
727 fail:
728     rdc_many_exit(krdc);
729     return (rc);
730 }
731
732 /*
733  * Move the set to a new group if possible
734  */
735 static int
736 change_group(rdc_k_info_t *krdc, int options)
737 {
738     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
739     rdc_u_info_t *utmp;
740     rdc_k_info_t *ktmp;
741     rdc_k_info_t *next;
742     char tmpq[NSC_MAXPATH];
743     int index;
744     int rc = -1;
745     rdc_group_t *group, *old_group;
746     nstthread_t *trc;
747
748     ASSERT(MUTEX_HELD(&rdc_conf_lock));
749
750     /*
751      * Look for matching group name, primary host name and secondary
752      * host name.
753      */
754
755     bzero(&tmpq, sizeof(tmpq));
756     rdc_many_enter(krdc);
757
758     old_group = krdc->group;
759     next = krdc->group_next;
760
761     if (RDC_IS_DISKQ(old_group)) { /* can't keep your own queue */
762         (void) strncpy(tmpq, urdc->disk_queue, NSC_MAXPATH);
763         bzero(urdc->disk_queue, sizeof(urdc->disk_queue));
764     }
765     for (index = 0; index < rdc_max_sets; index++) {
766         utmp = &rdc_u_info[index];
767         ktmp = &rdc_k_info[index];
768
769         if (ktmp == krdc)
770             continue;
771
772         if (urdc->group_name[0] == 0)
773             break;
774
775         if (!IS_CONFIGURED(ktmp))
776             continue;
777
778         if (strcmp(utmp->group_name, urdc->group_name,
779                 NSC_MAXPATH) != 0)
780             continue;
781         if (strcmp(utmp->primary_intf, urdc->primary_intf,
782                 MAX_RDC_HOST_SIZE) != 0)
783             goto bad;
784         if (strcmp(utmp->secondary_intf, urdc->secondary_intf,
785                 MAX_RDC_HOST_SIZE) != 0)
786             goto bad;
787     }

```

```

789         /* Group already exists, so add this set to the group */
791         if (((options & RDC_OPT_ASYNC) == 0) &&
792             ((ktmp->type_flag & RDC_ASYNCMODE) != 0)) {
793             /* Must be same mode as existing group members */
794             goto bad;
795         }
796         if (((options & RDC_OPT_ASYNC) != 0) &&
797             ((ktmp->type_flag & RDC_ASYNCMODE) == 0)) {
798             /* Must be same mode as existing group members */
799             goto bad;
800         }
802         ktmp->group->count++;
803         krdc->group = ktmp->group;
804         krdc->group_next = ktmp->group_next;
805         ktmp->group_next = krdc;
806         bzero(urdc->disk_queue, sizeof(urdc->disk_queue));
807         (void) strncpy(urdc->disk_queue, utmp->disk_queue, NSC_MAXPATH);
809         goto good;
810     }
812     /* This must be a new group */
813     group = rdc_newgroup();
814     krdc->group = group;
815     krdc->group_next = krdc;
817     trc = nst_create(_rdc_ioset, rdc_qfiller_thr, (void *)krdc, NST_SLEEP);
818     if (trc == NULL) {
819         rc = -1;
820         cmn_err(CE_NOTE, "!unable to create queue filler daemon");
821         goto bad;
822     }
824     if (urdc->disk_queue[0] == 0) {
825         krdc->group->flags |= RDC_MEMQUE;
826     } else {
827         krdc->group->flags |= RDC_DISKQUE;
828         if ((rc = rdc_enable_diskq(krdc) < 0)
829             goto bad;
830     }
831 good:
832     if (options & RDC_OPT_ASYNC) {
833         krdc->type_flag |= RDC_ASYNCMODE;
834         rdc_set_flags(urdc, RDC_ASYNC);
835     } else {
836         krdc->type_flag &= ~RDC_ASYNCMODE;
837         rdc_clr_flags(urdc, RDC_ASYNC);
838     }
840     old_group->count--;
841     if (!old_group->rdc_writer && old_group->count == 0) {
842         /* Group now empty, so destroy */
843         if (RDC_IS_DISKQ(old_group)) {
844             rdc_unintercept_diskq(old_group);
845             mutex_enter(&old_group->diskqmutex);
846             rdc_close_diskq(old_group);
847             mutex_exit(&old_group->diskqmutex);
848         }
850         mutex_enter(&old_group->ra_queue.net_qlock);
852         /*
853         * Assure the we've stopped and the flusher thread has not

```

```

854         * fallen back to sleep
855         */
856         if (old_group->ra_queue.qfill_sleeping != RDC_QFILL_DEAD) {
857             old_group->ra_queue.qfflags |= RDC_QFILLSTOP;
858             while (old_group->ra_queue.qfflags & RDC_QFILLSTOP) {
859                 if (old_group->ra_queue.qfill_sleeping ==
860                     RDC_QFILL_ASLEEP)
861                     cv_broadcast(&old_group->ra_queue.qfcv);
862                 mutex_exit(&old_group->ra_queue.net_qlock);
863                 delay(2);
864                 mutex_enter(&old_group->ra_queue.net_qlock);
865             }
866         }
867         mutex_exit(&old_group->ra_queue.net_qlock);
869         rdc_delgroup(old_group);
870         rdc_many_exit(krdc);
871         return (0);
872     }
874     /* Take this rdc structure off the old group list */
876     for (ktmp = next; ktmp->group_next != krdc; ktmp = ktmp->group_next)
877         ;
878     ktmp->group_next = next;
880     rdc_many_exit(krdc);
881     return (0);
883 bad:
884     /* Leave existing group status alone */
885     (void) strncpy(urdc->disk_queue, tmpq, NSC_MAXPATH);
886     rdc_many_exit(krdc);
887     return (rc);
888 }
891 /*
892 * Set flags for an rdc set, setting the group flags as necessary.
893 */
894 void
895 rdc_set_flags(rdc_u_info_t *urdc, int flags)
896 {
897     rdc_k_info_t *krdc = &rdc_k_info[urdc->index];
898     int vflags, sflags, bflags, ssflags;
900     DTRACE_PROBE2(rdc_set_flags, int, krdc->index, int, flags);
901     vflags = flags & RDC_VFLAGS;
902     sflags = flags & RDC_SFLAGS;
903     bflags = flags & RDC_BFLAGS;
904     ssflags = flags & RDC_SYNC_STATE_FLAGS;
906     if (vflags) {
907         /* normal volume flags */
908         ASSERT(MUTEX_HELD(&rdc_conf_lock) ||
909             MUTEX_HELD(&krdc->group->lock));
910         if (ssflags)
911             mutex_enter(&krdc->bmapmutex);
913         urdc->flags |= vflags;
915         if (ssflags)
916             mutex_exit(&krdc->bmapmutex);
917     }
919     if (sflags) {

```

```

920     /* Sync state flags that are protected by a different lock */
921     ASSERT(MUTEX_HELD(&rdc_many_lock));
922     urdc->sync_flags |= sflags;
923 }
924
925     if (bflags) {
926         /* Bmap state flags that are protected by a different lock */
927         ASSERT(MUTEX_HELD(&krdc->bmapmutex));
928         urdc->bmap_flags |= bflags;
929     }
930 }
931
932
933 /*
934 * Clear flags for an rdc set, clearing the group flags as necessary.
935 */
936 void
937 rdc_clr_flags(rdc_u_info_t *urdc, int flags)
938 {
939     rdc_k_info_t *krdc = &rdc_k_info[urdc->index];
940     int vflags, sflags, bflags;
941
942     DTRACE_PROBE2(rdc_clr_flags, int, krdc->index, int, flags);
943     vflags = flags & RDC_VFLAGS;
944     sflags = flags & RDC_SFLAGS;
945     bflags = flags & RDC_BFLAGS;
946
947     if (vflags) {
948         /* normal volume flags */
949         ASSERT(MUTEX_HELD(&rdc_conf_lock) ||
950             MUTEX_HELD(&krdc->group->lock));
951         urdc->flags &= ~vflags;
952     }
953
954     if (sflags) {
955         /* Sync state flags that are protected by a different lock */
956         ASSERT(MUTEX_HELD(&rdc_many_lock));
957         urdc->sync_flags &= ~sflags;
958     }
959
960     if (bflags) {
961         /* Bmap state flags that are protected by a different lock */
962         ASSERT(MUTEX_HELD(&krdc->bmapmutex));
963         urdc->bmap_flags &= ~bflags;
964     }
965 }
966
967
968 /*
969 * Get the flags for an rdc set.
970 */
971 int
972 rdc_get_vflags(rdc_u_info_t *urdc)
973 {
974     return (urdc->flags | urdc->sync_flags | urdc->bmap_flags);
975 }
976
977
978 /*
979 * Initialise flags for an rdc set.
980 */
981 static void
982 rdc_init_flags(rdc_u_info_t *urdc)
983 {

```

```

986     urdc->flags = 0;
987     urdc->mflags = 0;
988     urdc->sync_flags = 0;
989     urdc->bmap_flags = 0;
990 }
991
992
993 /*
994 * Set flags for a many group.
995 */
996 void
997 rdc_set_mflags(rdc_u_info_t *urdc, int flags)
998 {
999     rdc_k_info_t *krdc = &rdc_k_info[urdc->index];
1000    rdc_k_info_t *this = krdc;
1001
1002    ASSERT(!(flags & ~RDC_MFLAGS));
1003
1004    if (flags == 0)
1005        return;
1006
1007    ASSERT(MUTEX_HELD(&rdc_many_lock));
1008
1009    rdc_set_flags(urdc, flags); /* set flags on local urdc */
1010
1011    urdc->mflags |= flags;
1012    for (krdc = krdc->many_next; krdc != this; krdc = krdc->many_next) {
1013        urdc = &rdc_u_info[krdc->index];
1014        if (!IS_ENABLED(urdc))
1015            continue;
1016        urdc->mflags |= flags;
1017    }
1018 }
1019
1020
1021 /*
1022 * Clear flags for a many group.
1023 */
1024 void
1025 rdc_clr_mflags(rdc_u_info_t *urdc, int flags)
1026 {
1027     rdc_k_info_t *krdc = &rdc_k_info[urdc->index];
1028     rdc_k_info_t *this = krdc;
1029     rdc_u_info_t *utmp;
1030
1031     ASSERT(!(flags & ~RDC_MFLAGS));
1032
1033     if (flags == 0)
1034         return;
1035
1036     ASSERT(MUTEX_HELD(&rdc_many_lock));
1037
1038     rdc_clr_flags(urdc, flags); /* clear flags on local urdc */
1039
1040     /*
1041      * We must maintain the mflags based on the set of flags for
1042      * all the urdc's that are chained up.
1043      */
1044
1045     /*
1046      * First look through all the urdc's and remove bits from
1047      * the 'flags' variable that are in use elsewhere.
1048      */
1049
1050     for (krdc = krdc->many_next; krdc != this; krdc = krdc->many_next) {
1051         utmp = &rdc_u_info[krdc->index];

```

```

1052         if (!IS_ENABLED(utmp))
1053             continue;
1054         flags &= ~(rdc_get_vflags(utmp) & RDC_MFLAGS);
1055         if (flags == 0)
1056             break;
1057     }
1058
1059     /*
1060     * Now clear flags as necessary.
1061     */
1062
1063     if (flags != 0) {
1064         urdc->mflags &= ~flags;
1065         for (krdc = krdc->many_next; krdc != this;
1066             krdc = krdc->many_next) {
1067             utmp = &rdc_u_info[krdc->index];
1068             if (!IS_ENABLED(utmp))
1069                 continue;
1070             utmp->mflags &= ~flags;
1071         }
1072     }
1073 }
1074
1075
1076 int
1077 rdc_get_mflags(rdc_u_info_t *urdc)
1078 {
1079     return (urdc->mflags);
1080 }
1081
1082
1083 void
1084 rdc_set_flags_log(rdc_u_info_t *urdc, int flags, char *why)
1085 {
1086     DTRACE_PROBE2(rdc_set_flags_log, int, urdc->index, int, flags);
1087
1088     rdc_set_flags(urdc, flags);
1089
1090     if (why == NULL)
1091         return;
1092
1093     if (flags & RDC_LOGGING)
1094         cmn_err(CE_NOTE, "!sndr: %s:%s entered logging mode: %s",
1095             urdc->secondary.intf, urdc->secondary.file, why);
1096     if (flags & RDC_VOL_FAILED)
1097         cmn_err(CE_NOTE, "!sndr: %s:%s volume failed: %s",
1098             urdc->secondary.intf, urdc->secondary.file, why);
1099     if (flags & RDC_BMP_FAILED)
1100         cmn_err(CE_NOTE, "!sndr: %s:%s bitmap failed: %s",
1101             urdc->secondary.intf, urdc->secondary.file, why);
1102 }
1103 /*
1104 * rdc_lor(source, dest, len)
1105 * logically OR memory pointed to by source and dest, copying result into dest.
1106 */
1107 void
1108 rdc_lor(const uchar_t *source, uchar_t *dest, int len)
1109 {
1110     int i;
1111
1112     if (source == NULL)
1113         return;
1114
1115     for (i = 0; i < len; i++)
1116         *dest++ |= *source++;
1117 }

```

```

1120 static int
1121 check_filesize(int index, spcs_s_info_t kstatus)
1122 {
1123     uint64_t remote_size;
1124     char tmp1[16], tmp2[16];
1125     rdc_u_info_t *urdc = &rdc_u_info[index];
1126     int status;
1127
1128     status = rdc_net_getsize(index, &remote_size);
1129     if (status) {
1130         (void) spcs_s_inttostring(status, tmp1, sizeof (tmp1), 0);
1131         spcs_s_add(kstatus, RDC_EGETSIZE, urdc->secondary.intf,
1132             urdc->secondary.file, tmp1);
1133         (void) rdc_net_state(index, CCIO_ENABLELOG);
1134         return (RDC_EGETSIZE);
1135     }
1136     if (remote_size < (unsigned long long)urdc->volume_size) {
1137         (void) spcs_s_inttostring(
1138             urdc->volume_size, tmp1, sizeof (tmp1), 0);
1139         /*
1140         * Cheat, and covert to int, until we have
1141         * spcs_s_unsignedlonginttostring().
1142         */
1143         status = (int)remote_size;
1144         (void) spcs_s_inttostring(status, tmp2, sizeof (tmp2), 0);
1145         spcs_s_add(kstatus, RDC_ESIZE, urdc->primary.intf,
1146             urdc->primary.file, tmp1, urdc->secondary.intf,
1147             urdc->secondary.file, tmp2);
1148         (void) rdc_net_state(index, CCIO_ENABLELOG);
1149         return (RDC_ESIZE);
1150     }
1151     return (0);
1152 }
1153
1154
1155 static void
1156 rdc_volume_update_svc(intptr_t arg)
1157 {
1158     rdc_update_t *update = (rdc_update_t *)arg;
1159     rdc_k_info_t *krdc;
1160     rdc_k_info_t *this;
1161     rdc_u_info_t *urdc;
1162     struct net_bdata6 bd;
1163     int index;
1164     int rc;
1165
1166 #ifdef DEBUG_IUPDATE
1167     cmn_err(CE_NOTE, "!SNDR received update request for %s",
1168         update->volume);
1169 #endif
1170
1171     if ((update->protocol != RDC_SVC_ONRETURN) &&
1172         (update->protocol != RDC_SVC_VOL_ENABLED)) {
1173         /* don't understand what the client intends to do */
1174         update->denied = 1;
1175         spcs_s_add(update->status, RDC_EVERSION);
1176         return;
1177     }
1178
1179     index = rdc_lookup_enabled(update->volume, 0);
1180     if (index < 0)
1181         return;
1182
1183     /*

```

```

1184     * warn II that this volume is in use by sndr so
1185     * II can validate the sizes of the master vs shadow
1186     * and avoid trouble later down the line with
1187     * size mis-matches between urdc->volume_size and
1188     * what is returned from nsc_partsize() which may
1189     * be the size of the master when replicating the shadow
1190     */
1191     if (update->protocol == RDC_SVC_VOL_ENABLED) {
1192         if (index >= 0)
1193             update->denied = 1;
1194         return;
1195     }
1197     krdc = &rdc_k_info[index];
1198     urdc = &rdc_u_info[index];
1199     this = krdc;
1201     do {
1202         if (!(rdc_get_vflags(urdc) & RDC_LOGGING)) {
1203 #ifndef DEBUG_IIUPDATE
1204             cmn_err(CE_NOTE, "!SNDR refused update request for %s",
1205                 update->volume);
1206 #endif
1207             update->denied = 1;
1208             spcs_s_add(update->status, RDC_EMIRRORUP);
1209             return;
1210         }
1211         /* 1->many - all must be logging */
1212         if (IS_MANY(krdc) && IS_STATE(urdc, RDC_PRIMARY)) {
1213             rdc_many_enter(krdc);
1214             for (krdc = krdc->many_next; krdc != this;
1215                 krdc = krdc->many_next) {
1216                 urdc = &rdc_u_info[krdc->index];
1217                 if (!IS_ENABLED(urdc))
1218                     continue;
1219                 break;
1220             }
1221             rdc_many_exit(krdc);
1222         }
1223     } while (krdc != this);
1225 #ifndef DEBUG_IIUPDATE
1226     cmn_err(CE_NOTE, "!SNDR allowed update request for %s", update->volume);
1227 #endif
1228     urdc = &rdc_u_info[krdc->index];
1229     do {
1231         bd.size = min(krdc->bitmap_size, (nsc_size_t)update->size);
1232         bd.data.data_val = (char *)update->bitmap;
1233         bd.offset = 0;
1234         bd.cd = index;
1236         if ((rc = RDC_OR_BITMAP(&bd)) != 0) {
1237             update->denied = 1;
1238             spcs_s_add(update->status, rc);
1239             return;
1240         }
1241         urdc = &rdc_u_info[index];
1242         urdc->bits_set = RDC_COUNT_BITMAP(krdc);
1243         if (IS_MANY(krdc) && IS_STATE(urdc, RDC_PRIMARY)) {
1244             rdc_many_enter(krdc);
1245             for (krdc = krdc->many_next; krdc != this;
1246                 krdc = krdc->many_next) {
1247                 index = krdc->index;
1248                 if (!IS_ENABLED(urdc))
1249                     continue;

```

```

1250         break;
1251     }
1252     rdc_many_exit(krdc);
1253 }
1254 } while (krdc != this);
1257 /* II (or something else) has updated us, so no need for a sync */
1258 if (rdc_get_vflags(urdc) & (RDC_SYNC_NEEDED | RDC_RSYNC_NEEDED)) {
1259     rdc_many_enter(krdc);
1260     rdc_clr_flags(urdc, RDC_SYNC_NEEDED | RDC_RSYNC_NEEDED);
1261     rdc_many_exit(krdc);
1262 }
1264 if (krdc->bitmap_write > 0)
1265     (void) rdc_write_bitmap(krdc);
1266 }
1269 /*
1270  * rdc_check()
1271  *
1272  * Return 0 if the set is configured, enabled and the supplied
1273  * addressing information matches the in-kernel config, otherwise
1274  * return 1.
1275  */
1276 static int
1277 rdc_check(rdc_k_info_t *krdc, rdc_set_t *rdc_set)
1278 {
1279     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
1281     ASSERT(MUTEX_HELD(&krdc->group->lock));
1283     if (!IS_ENABLED(urdc))
1284         return (1);
1286     if (strcmp(urdc->primary.file, rdc_set->primary.file,
1287         NSC_MAXPATH) != 0) {
1288 #ifndef DEBUG
1289         cmn_err(CE_WARN, "!rdc_check: primary file mismatch %s vs %s",
1290             urdc->primary.file, rdc_set->primary.file);
1291 #endif
1292         return (1);
1293     }
1295     if (rdc_set->primary.addr.len != 0 &&
1296         bcmp(urdc->primary.addr.buf, rdc_set->primary.addr.buf,
1297             urdc->primary.addr.len) != 0) {
1298 #ifndef DEBUG
1299         cmn_err(CE_WARN, "!rdc_check: primary address mismatch for %s",
1300             urdc->primary.file);
1301 #endif
1302         return (1);
1303     }
1305     if (strcmp(urdc->secondary.file, rdc_set->secondary.file,
1306         NSC_MAXPATH) != 0) {
1307 #ifndef DEBUG
1308         cmn_err(CE_WARN, "!rdc_check: secondary file mismatch %s vs %s",
1309             urdc->secondary.file, rdc_set->secondary.file);
1310 #endif
1311         return (1);
1312     }
1314     if (rdc_set->secondary.addr.len != 0 &&
1315         bcmp(urdc->secondary.addr.buf, rdc_set->secondary.addr.buf,

```

```

1316         urdc->secondary.addr.len) != 0) {
1317 #ifdef DEBUG
1318         cmn_err(CE_WARN, "!rdc_check: secondary addr mismatch for %s",
1319             urdc->secondary.file);
1320 #endif
1321         return (1);
1322     }
1324     return (0);
1325 }

1328 /*
1329  * Lookup enabled sets for a bitmap match
1330  */

1332 int
1333 rdc_lookup_bitmap(char *pathname)
1334 {
1335     rdc_u_info_t *urdc;
1336 #ifdef DEBUG
1337     rdc_k_info_t *krdc;
1338 #endif
1339     int index;

1341     for (index = 0; index < rdc_max_sets; index++) {
1342         urdc = &rdc_u_info[index];
1343 #ifdef DEBUG
1344         krdc = &rdc_k_info[index];
1345 #endif
1346         ASSERT(krdc->index == index);
1347         ASSERT(urdc->index == index);

1349         if (!IS_ENABLED(urdc))
1350             continue;

1352         if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
1353             if (strncmp(pathname, urdc->primary.bitmap,
1354                 NSC_MAXPATH) == 0)
1355                 return (index);
1356         } else {
1357             if (strncmp(pathname, urdc->secondary.bitmap,
1358                 NSC_MAXPATH) == 0)
1359                 return (index);
1360         }
1361     }

1363     return (-1);
1364 }

1367 /*
1368  * Translate a pathname to index into rdc_k_info[].
1369  * Returns first match that is enabled.
1370  */

1372 int
1373 rdc_lookup_enabled(char *pathname, int allow_disabling)
1374 {
1375     rdc_u_info_t *urdc;
1376     rdc_k_info_t *krdc;
1377     int index;

1379 restart:
1380     for (index = 0; index < rdc_max_sets; index++) {
1381         urdc = &rdc_u_info[index];

```

```

1382         krdc = &rdc_k_info[index];

1384         ASSERT(krdc->index == index);
1385         ASSERT(urdc->index == index);

1387         if (!IS_ENABLED(urdc))
1388             continue;

1390         if (allow_disabling == 0 && krdc->type_flag & RDC_UNREGISTER)
1391             continue;

1393         if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
1394             if (strncmp(pathname, urdc->primary.file,
1395                 NSC_MAXPATH) == 0)
1396                 return (index);
1397         } else {
1398             if (strncmp(pathname, urdc->secondary.file,
1399                 NSC_MAXPATH) == 0)
1400                 return (index);
1401         }
1402     }

1404     if (allow_disabling == 0) {
1405         /* None found, or only a disabling one found, so try again */
1406         allow_disabling = 1;
1407         goto restart;
1408     }

1410     return (-1);
1411 }

1414 /*
1415  * Translate a pathname to index into rdc_k_info[].
1416  * Returns first match that is configured.
1417  * Used by enable & resume code.
1418  * Must be called with rdc_conf_lock held.
1419  */

1422 int
1423 rdc_lookup_configured(char *pathname)
1424 {
1425     rdc_u_info_t *urdc;
1426     rdc_k_info_t *krdc;
1427     int index;

1429     ASSERT(MUTEX_HELD(&rdc_conf_lock));

1431     for (index = 0; index < rdc_max_sets; index++) {
1432         urdc = &rdc_u_info[index];
1433         krdc = &rdc_k_info[index];

1435         ASSERT(krdc->index == index);
1436         ASSERT(urdc->index == index);

1438         if (!IS_CONFIGURED(krdc))
1439             continue;

1441         if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
1442             if (strncmp(pathname, urdc->primary.file,
1443                 NSC_MAXPATH) == 0)
1444                 return (index);
1445         } else {
1446             if (strncmp(pathname, urdc->secondary.file,
1447                 NSC_MAXPATH) == 0)

```

```

1448         return (index);
1449     }
1450 }

1452     return (-1);
1453 }

1456 /*
1457 * Looks up a configured set with matching secondary interface:volume
1458 * to check for illegal many-to-one volume configs. To be used during
1459 * enable and resume processing.
1460 *
1461 * Must be called with rdc_conf_lock held.
1462 */

1464 static int
1465 rdc_lookup_many2one(rdc_set_t *rdc_set)
1466 {
1467     rdc_u_info_t *urdc;
1468     rdc_k_info_t *krdc;
1469     int index;

1471     ASSERT(MUTEX_HELD(&rdc_conf_lock));

1473     for (index = 0; index < rdc_max_sets; index++) {
1474         urdc = &rdc_u_info[index];
1475         krdc = &rdc_k_info[index];

1477         if (!IS_CONFIGURED(krdc))
1478             continue;

1480         if (strcmp(urdc->secondary.file,
1481                 rdc_set->secondary.file, NSC_MAXPATH) != 0)
1482             continue;
1483         if (strcmp(urdc->secondary.intf,
1484                 rdc_set->secondary.intf, MAX_RDC_HOST_SIZE) != 0)
1485             continue;

1487         break;
1488     }

1490     if (index < rdc_max_sets)
1491         return (index);
1492     else
1493         return (-1);
1494 }

1497 /*
1498 * Looks up an rdc set to check if it is already configured, to be used from
1499 * functions called from the config ioctl where the interface names can be
1500 * used for comparison.
1501 *
1502 * Must be called with rdc_conf_lock held.
1503 */

1505 int
1506 rdc_lookup_byname(rdc_set_t *rdc_set)
1507 {
1508     rdc_u_info_t *urdc;
1509     rdc_k_info_t *krdc;
1510     int index;

1512     ASSERT(MUTEX_HELD(&rdc_conf_lock));

```

```

1514         for (index = 0; index < rdc_max_sets; index++) {
1515             urdc = &rdc_u_info[index];
1516             krdc = &rdc_k_info[index];

1518             ASSERT(krdc->index == index);
1519             ASSERT(urdc->index == index);

1521             if (!IS_CONFIGURED(krdc))
1522                 continue;

1524             if (strcmp(urdc->primary.file, rdc_set->primary.file,
1525                     NSC_MAXPATH) != 0)
1526                 continue;
1527             if (strcmp(urdc->primary.intf, rdc_set->primary.intf,
1528                     MAX_RDC_HOST_SIZE) != 0)
1529                 continue;
1530             if (strcmp(urdc->secondary.file, rdc_set->secondary.file,
1531                     NSC_MAXPATH) != 0)
1532                 continue;
1533             if (strcmp(urdc->secondary.intf, rdc_set->secondary.intf,
1534                     MAX_RDC_HOST_SIZE) != 0)
1535                 continue;

1537             break;
1538         }

1540         if (index < rdc_max_sets)
1541             return (index);
1542         else
1543             return (-1);
1544     }

1546 /*
1547 * Looks up a secondary hostname and device, to be used from
1548 * functions called from the config ioctl where the interface names can be
1549 * used for comparison.
1550 *
1551 * Must be called with rdc_conf_lock held.
1552 */

1554 int
1555 rdc_lookup_byhostdev(char *intf, char *file)
1556 {
1557     rdc_u_info_t *urdc;
1558     rdc_k_info_t *krdc;
1559     int index;

1561     ASSERT(MUTEX_HELD(&rdc_conf_lock));

1563     for (index = 0; index < rdc_max_sets; index++) {
1564         urdc = &rdc_u_info[index];
1565         krdc = &rdc_k_info[index];

1567         ASSERT(krdc->index == index);
1568         ASSERT(urdc->index == index);

1570         if (!IS_CONFIGURED(krdc))
1571             continue;

1573         if (strcmp(urdc->secondary.file, file,
1574                 NSC_MAXPATH) != 0)
1575             continue;
1576         if (strcmp(urdc->secondary.intf, intf,
1577                 MAX_RDC_HOST_SIZE) != 0)
1578             continue;
1579         break;

```

```

1580     }
1582     if (index < rdc_max_sets)
1583         return (index);
1584     else
1585         return (-1);
1586 }

1589 /*
1590  * Looks up an rdc set to see if it is currently enabled, to be used on the
1591  * server so that the interface addresses must be used for comparison, as
1592  * the interface names may differ from those used on the client.
1593  *
1594  */

1596 int
1597 rdc_lookup_byaddr(rdc_set_t *rdc_set)
1598 {
1599     rdc_u_info_t *urdc;
1600 #ifdef DEBUG
1601     rdc_k_info_t *krdc;
1602 #endif
1603     int index;

1605     for (index = 0; index < rdc_max_sets; index++) {
1606         urdc = &rdc_u_info[index];
1607 #ifdef DEBUG
1608         krdc = &rdc_k_info[index];
1609 #endif
1610         ASSERT(krdc->index == index);
1611         ASSERT(urdc->index == index);

1613         if (!IS_ENABLED(urdc))
1614             continue;

1616         if (strcmp(urdc->primary.file, rdc_set->primary.file) != 0)
1617             continue;

1619         if (strcmp(urdc->secondary.file, rdc_set->secondary.file) != 0)
1620             continue;

1622         if (bcmp(urdc->primary.addr.buf, rdc_set->primary.addr.buf,
1623                urdc->primary.addr.len) != 0) {
1624             continue;
1625         }

1627         if (bcmp(urdc->secondary.addr.buf, rdc_set->secondary.addr.buf,
1628                urdc->secondary.addr.len) != 0) {
1629             continue;
1630         }

1632         break;
1633     }

1635     if (index < rdc_max_sets)
1636         return (index);
1637     else
1638         return (-1);
1639 }

1642 /*
1643  * Return index of first multihop or 1-to-many
1644  * Behavior controlled by setting ismany.
1645  * ismany TRUE (one-to-many)

```

```

1646  * ismany FALSE (multihops)
1647  *
1648  */
1649 static int
1650 rdc_lookup_multimany(rdc_k_info_t *krdc, const int ismany)
1651 {
1652     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
1653     rdc_u_info_t *utmp;
1654     rdc_k_info_t *ktmp;
1655     char *pathname;
1656     int index;
1657     int role;

1659     ASSERT(MUTEX_HELD(&rdc_conf_lock));
1660     ASSERT(MUTEX_HELD(&rdc_many_lock));

1662     if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
1663         /* this host is the primary of the krdc set */
1664         pathname = urdc->primary.file;
1665         if (ismany) {
1666             /*
1667              * 1-many sets are linked by primary :
1668              * look for matching primary on this host
1669              */
1670             role = RDC_PRIMARY;
1671         } else {
1672             /*
1673              * multihop sets link primary to secondary :
1674              * look for matching secondary on this host
1675              */
1676             role = 0;
1677         }
1678     } else {
1679         /* this host is the secondary of the krdc set */
1680         pathname = urdc->secondary.file;
1681         if (ismany) {
1682             /*
1683              * 1-many sets are linked by primary, so if
1684              * this host is the secondary of the set this
1685              * cannot require 1-many linkage.
1686              */
1687             return (-1);
1688         } else {
1689             /*
1690              * multihop sets link primary to secondary :
1691              * look for matching primary on this host
1692              */
1693             role = RDC_PRIMARY;
1694         }
1695     }

1697     for (index = 0; index < rdc_max_sets; index++) {
1698         utmp = &rdc_u_info[index];
1699         ktmp = &rdc_k_info[index];

1701         if (!IS_CONFIGURED(ktmp)) {
1702             continue;
1703         }

1705         if (role == RDC_PRIMARY) {
1706             /*
1707              * Find a primary that is this host and is not
1708              * krdc but shares the same data volume as krdc.
1709              */
1710             if ((rdc_get_vflags(utmp) & RDC_PRIMARY) &&
1711                 strcmp(utmp->primary.file, pathname,

```



```

1712         NSC_MAXPATH) == 0 && (krdc != ktmp)) {
1713             break;
1714         } else {
1715             /*
1716             * Find a secondary that is this host and is not
1717             * krdc but shares the same data volume as krdc.
1718             */
1719             if (!(rdc_get_vflags(utmp) & RDC_PRIMARY) &&
1720                 strcmp(utmp->secondary.file, pathname,
1721                     NSC_MAXPATH) == 0 && (krdc != ktmp)) {
1722                 break;
1723             }
1724         }
1725     }
1726 }

1728     if (index < rdc_max_sets)
1729         return (index);
1730     else
1731         return (-1);
1732 }

1734 /*
1735 * Returns secondary match that is configured.
1736 *
1737 * Used by enable & resume code.
1738 * Must be called with rdc_conf_lock held.
1739 */

1741 static int
1742 rdc_lookup_secondary(char *pathname)
1743 {
1744     rdc_u_info_t *urdc;
1745     rdc_k_info_t *krdc;
1746     int index;

1748     ASSERT(MUTEX_HELD(&rdc_conf_lock));

1750     for (index = 0; index < rdc_max_sets; index++) {
1751         urdc = &rdc_u_info[index];
1752         krdc = &rdc_k_info[index];

1754         ASSERT(krdc->index == index);
1755         ASSERT(urdc->index == index);

1757         if (!IS_CONFIGURED(krdc))
1758             continue;

1760         if (!IS_STATE(urdc, RDC_PRIMARY)) {
1761             if (strcmp(pathname, urdc->secondary.file,
1762                 NSC_MAXPATH) == 0)
1763                 return (index);
1764         }
1765     }

1767     return (-1);
1768 }

1771 static nsc_fd_t *
1772 rdc_open_direct(rdc_k_info_t *krdc)
1773 {
1774     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
1775     int rc;

1777     if (krdc->remote_fd == NULL)

```

```

1778         krdc->remote_fd = nsc_open(urdc->direct_file,
1779             NSC_RDCHR_ID|NSC_DEVICE|NSC_RDWR, 0, 0, &rc);
1780         return (krdc->remote_fd);
1781     }

1783 static void
1784 rdc_close_direct(rdc_k_info_t *krdc)
1785 {
1786     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];

1788     urdc->direct_file[0] = 0;
1789     if (krdc->remote_fd) {
1790         if (nsc_close(krdc->remote_fd) == 0) {
1791             krdc->remote_fd = NULL;
1792         }
1793     }
1794 }

1797 #ifdef DEBUG_MANY
1798 static void
1799 print_many(rdc_k_info_t *start)
1800 {
1801     rdc_k_info_t *p = start;
1802     rdc_u_info_t *q = &rdc_u_info[p->index];

1804     do {
1805         cmn_err(CE_CONT, "!krdc %p, %s %s (many_nxt %p multi_nxt %p)\n",
1806             p, q->primary.file, q->secondary.file, p->many_next,
1807             p->multi_next);
1808         delay(10);
1809         p = p->many_next;
1810         q = &rdc_u_info[p->index];
1811     } while (p && p != start);
1812 }
1813 #endif /* DEBUG_MANY */

1816 static int
1817 add_to_multi(rdc_k_info_t *krdc)
1818 {
1819     rdc_u_info_t *urdc;
1820     rdc_k_info_t *ktmp;
1821     rdc_u_info_t *utmp;
1822     int mindex;
1823     int domulti;

1825     urdc = &rdc_u_info[krdc->index];

1827     ASSERT(MUTEX_HELD(&rdc_conf_lock));
1828     ASSERT(MUTEX_HELD(&rdc_many_lock));

1830     /* Now find companion krdc */
1831     mindex = rdc_lookup_multimany(krdc, FALSE);

1833 #ifdef DEBUG_MANY
1834     cmn_err(CE_NOTE,
1835         "!add_to_multi: lookup_multimany: mindex %d prim %s sec %s",
1836         mindex, urdc->primary.file, urdc->secondary.file);
1837 #endif

1839     if (mindex >= 0) {
1840         ktmp = &rdc_k_info[mindex];
1841         utmp = &rdc_u_info[mindex];

1843         domulti = 1;

```

```

1845         if ((rdc_get_vflags(urdc) & RDC_PRIMARY) &&
1846             ktmp->multi_next != NULL) {
1847             /*
1848              * We are adding a new primary to a many
1849              * group that is the target of a multihop, just
1850              * ignore it since we are linked in elsewhere.
1851              */
1852             domulti = 0;
1853         }
1854
1855         if (domulti) {
1856             if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
1857                 /* Is previous leg using direct file I/O? */
1858                 if (utmp->direct_file[0] != 0) {
1859                     /* It is, so cannot proceed */
1860                     return (-1);
1861                 }
1862             } else {
1863                 /* Is this leg using direct file I/O? */
1864                 if (urdc->direct_file[0] != 0) {
1865                     /* It is, so cannot proceed */
1866                     return (-1);
1867                 }
1868             }
1869             krdc->multi_next = ktmp;
1870             ktmp->multi_next = krdc;
1871         } else {
1872             krdc->multi_next = NULL;
1873         }
1874 #ifdef DEBUG_MANY
1875         cmn_err(CE_NOTE, "!add_to_multi: NULL multi_next index %d",
1876             krdc->index);
1877 #endif
1878     }
1879
1880     return (0);
1881 }
1882
1883 /*
1884 * Add a new set to the circular list of 1-to-many primaries and chain
1885 * up any multihop as well.
1886 */
1887 static int
1888 add_to_many(rdc_k_info_t *krdc)
1889 {
1890     rdc_k_info_t *okrdc;
1891     int oindex;
1892
1893     ASSERT(MUTEX_HELD(&rdc_conf_lock));
1894
1895     rdc_many_enter(krdc);
1896
1897     if (add_to_multi(krdc) < 0) {
1898         rdc_many_exit(krdc);
1899         return (-1);
1900     }
1901
1902     oindex = rdc_lookup_multimany(krdc, TRUE);
1903     if (oindex < 0) {
1904 #ifdef DEBUG_MANY
1905         print_many(krdc);
1906 #endif
1907         rdc_many_exit(krdc);
1908         return (0);
1909     }

```

```

1910     }
1911
1912     okrdc = &rdc_k_info[oindex];
1913
1914 #ifdef DEBUG_MANY
1915     print_many(okrdc);
1916 #endif
1917     krdc->many_next = okrdc->many_next;
1918     okrdc->many_next = krdc;
1919
1920 #ifdef DEBUG_MANY
1921     print_many(okrdc);
1922 #endif
1923     rdc_many_exit(krdc);
1924     return (0);
1925 }
1926
1927 /*
1928 * Remove a set from the circular list of 1-to-many primaries.
1929 */
1930 static void
1931 remove_from_many(rdc_k_info_t *old)
1932 {
1933     rdc_u_info_t *uold = &rdc_u_info[old->index];
1934     rdc_k_info_t *p, *q;
1935
1936     ASSERT(MUTEX_HELD(&rdc_conf_lock));
1937
1938     rdc_many_enter(old);
1939
1940 #ifdef DEBUG_MANY
1941     cmn_err(CE_NOTE, "!rdc: before remove_from_many");
1942     print_many(old);
1943 #endif
1944
1945     if (old->many_next == old) {
1946         /* remove from multihop */
1947         if ((q = old->multi_next) != NULL) {
1948             ASSERT(q->multi_next == old);
1949             q->multi_next = NULL;
1950             old->multi_next = NULL;
1951         }
1952
1953         rdc_many_exit(old);
1954         return;
1955     }
1956
1957     /* search */
1958     for (p = old->many_next; p->many_next != old; p = p->many_next)
1959         ;
1960
1961     p->many_next = old->many_next;
1962     old->many_next = old;
1963
1964     if ((q = old->multi_next) != NULL) {
1965         /*
1966          * old was part of a multihop, so switch multi pointers
1967          * to someone remaining on the many chain
1968          */
1969         ASSERT(p->multi_next == NULL);
1970
1971         q->multi_next = p;
1972         p->multi_next = q;
1973         old->multi_next = NULL;
1974     }
1975 }

```

```

1977 #ifndef DEBUG_MANY
1978     if (p == old) {
1979         cmn_err(CE_NOTE, "!rdc: after remove_from_many empty");
1980     } else {
1981         cmn_err(CE_NOTE, "!rdc: after remove_from_many");
1982         print_many(p);
1983     }
1984 #endif

1986     rdc_clr_mflags(&rdc_u_info[p->index],
1987                 (rdc_get_vflags(uold) & RDC_MFLAGS));

1989     rdc_many_exit(old);
1990 }

1993 static int
1994 _rdc_enable(rdc_set_t *rdc_set, int options, spcs_s_info_t kstatus)
1995 {
1996     int index;
1997     char *rhost;
1998     struct netbuf *addrp;
1999     rdc_k_info_t *krdc;
2000     rdc_u_info_t *urdc;
2001     rdc_srv_t *svp = NULL;
2002     char *local_file;
2003     char *local_bitmap;
2004     char *diskq;
2005     int rc;
2006     nsc_size_t maxfbas;
2007     rdc_group_t *grp;

2009     if ((rdc_set->primary.intf[0] == 0) ||
2010         (rdc_set->primary.addr.len == 0) ||
2011         (rdc_set->primary.file[0] == 0) ||
2012         (rdc_set->primary.bitmap[0] == 0) ||
2013         (rdc_set->secondary.intf[0] == 0) ||
2014         (rdc_set->secondary.addr.len == 0) ||
2015         (rdc_set->secondary.file[0] == 0) ||
2016         (rdc_set->secondary.bitmap[0] == 0)) {
2017         spcs_s_add(kstatus, RDC_EEMPTY);
2018         return (RDC_EEMPTY);
2019     }

2021     /* Next check there aren't any enabled rdc sets which match. */

2023     mutex_enter(&rdc_conf_lock);

2025     if (rdc_lookup_byname(rdc_set) >= 0) {
2026         mutex_exit(&rdc_conf_lock);
2027         spcs_s_add(kstatus, RDC_EENABLED, rdc_set->primary.intf,
2028                 rdc_set->primary.file, rdc_set->secondary.intf,
2029                 rdc_set->secondary.file);
2030         return (RDC_EENABLED);
2031     }

2033     if (rdc_lookup_many2one(rdc_set) >= 0) {
2034         mutex_exit(&rdc_conf_lock);
2035         spcs_s_add(kstatus, RDC_EMANY2ONE, rdc_set->primary.intf,
2036                 rdc_set->primary.file, rdc_set->secondary.intf,
2037                 rdc_set->secondary.file);
2038         return (RDC_EMANY2ONE);
2039     }

2041     if (rdc_set->netconfig->knc_proto == NULL) {

```

```

2042         mutex_exit(&rdc_conf_lock);
2043         spcs_s_add(kstatus, RDC_ENETCONFIG);
2044         return (RDC_ENETCONFIG);
2045     }

2047     if (rdc_set->primary.addr.len == 0) {
2048         mutex_exit(&rdc_conf_lock);
2049         spcs_s_add(kstatus, RDC_ENETBUF, rdc_set->primary.file);
2050         return (RDC_ENETBUF);
2051     }

2053     if (rdc_set->secondary.addr.len == 0) {
2054         mutex_exit(&rdc_conf_lock);
2055         spcs_s_add(kstatus, RDC_ENETBUF, rdc_set->secondary.file);
2056         return (RDC_ENETBUF);
2057     }

2059     /* Check that the local data volume isn't in use as a bitmap */
2060     if (options & RDC_OPT_PRIMARY)
2061         local_file = rdc_set->primary.file;
2062     else
2063         local_file = rdc_set->secondary.file;
2064     if (rdc_lookup_bitmap(local_file) >= 0) {
2065         mutex_exit(&rdc_conf_lock);
2066         spcs_s_add(kstatus, RDC_EVOLINUSE, local_file);
2067         return (RDC_EVOLINUSE);
2068     }

2070     /* check that the secondary data volume isn't in use */
2071     if (!(options & RDC_OPT_PRIMARY)) {
2072         local_file = rdc_set->secondary.file;
2073         if (rdc_lookup_secondary(local_file) >= 0) {
2074             mutex_exit(&rdc_conf_lock);
2075             spcs_s_add(kstatus, RDC_EVOLINUSE, local_file);
2076             return (RDC_EVOLINUSE);
2077         }
2078     }

2080     /* check that the local data vol is not in use as a diskqueue */
2081     if (options & RDC_OPT_PRIMARY) {
2082         if (rdc_lookup_diskq(rdc_set->primary.file) >= 0) {
2083             mutex_exit(&rdc_conf_lock);
2084             spcs_s_add(kstatus,
2085                     RDC_EVOLINUSE, rdc_set->primary.file);
2086             return (RDC_EVOLINUSE);
2087         }
2088     }

2090     /* Check that the bitmap isn't in use as a data volume */
2091     if (options & RDC_OPT_PRIMARY)
2092         local_bitmap = rdc_set->primary.bitmap;
2093     else
2094         local_bitmap = rdc_set->secondary.bitmap;
2095     if (rdc_lookup_configured(local_bitmap) >= 0) {
2096         mutex_exit(&rdc_conf_lock);
2097         spcs_s_add(kstatus, RDC_EBMPINUSE, local_bitmap);
2098         return (RDC_EBMPINUSE);
2099     }

2101     /* Check that the bitmap isn't already in use as a bitmap */
2102     if (rdc_lookup_bitmap(local_bitmap) >= 0) {
2103         mutex_exit(&rdc_conf_lock);
2104         spcs_s_add(kstatus, RDC_EBMPINUSE, local_bitmap);
2105         return (RDC_EBMPINUSE);
2106     }

```

```

2108 /* check that the diskq (if here) is not in use */
2109 diskq = rdc_set->disk_queue;
2110 if (diskq[0] && rdc_diskq_inuse(rdc_set, diskq)) {
2111     mutex_exit(&rdc_conf_lock);
2112     spcs_s_add(kstatus, RDC_EDISKQINUSE, diskq);
2113     return (RDC_EDISKQINUSE);
2114 }

2117 /* Set urdc->volume_size */
2118 index = rdc_dev_open(rdc_set, options);
2119 if (index < 0) {
2120     mutex_exit(&rdc_conf_lock);
2121     if (options & RDC_OPT_PRIMARY)
2122         spcs_s_add(kstatus, RDC_EOPEN, rdc_set->primary.intf,
2123                 rdc_set->primary.file);
2124     else
2125         spcs_s_add(kstatus, RDC_EOPEN, rdc_set->secondary.intf,
2126                 rdc_set->secondary.file);
2127     return (RDC_EOPEN);
2128 }

2130 urdc = &rdc_u_info[index];
2131 krdc = &rdc_k_info[index];

2133 /* copy relevant parts of rdc_set to urdc field by field */

2135 (void) strncpy(urdc->primary.intf, rdc_set->primary.intf,
2136             MAX_RDC_HOST_SIZE);
2137 (void) strncpy(urdc->secondary.intf, rdc_set->secondary.intf,
2138             MAX_RDC_HOST_SIZE);

2140 (void) strncpy(urdc->group_name, rdc_set->group_name, NSC_MAXPATH);
2141 (void) strncpy(urdc->disk_queue, rdc_set->disk_queue, NSC_MAXPATH);

2143 dup_rdc_netbuf(&rdc_set->primary.addr, &urdc->primary.addr);
2144 (void) strncpy(urdc->primary.file, rdc_set->primary.file, NSC_MAXPATH);
2145 (void) strncpy(urdc->primary.bitmap, rdc_set->primary.bitmap,
2146             NSC_MAXPATH);

2148 dup_rdc_netbuf(&rdc_set->secondary.addr, &urdc->secondary.addr);
2149 (void) strncpy(urdc->secondary.file, rdc_set->secondary.file,
2150             NSC_MAXPATH);
2151 (void) strncpy(urdc->secondary.bitmap, rdc_set->secondary.bitmap,
2152             NSC_MAXPATH);

2154 urdc->setid = rdc_set->setid;

2156 /*
2157  * before we try to add to group, or create one, check out
2158  * if we are doing the wrong thing with the diskq
2159  */

2161 if (urdc->disk_queue[0] && (options & RDC_OPT_SYNC)) {
2162     mutex_exit(&rdc_conf_lock);
2163     rdc_dev_close(krdc);
2164     spcs_s_add(kstatus, RDC_EQWRONGMODE);
2165     return (RDC_EQWRONGMODE);
2166 }

2168 if ((rc = add_to_group(krdc, options, RDC_CMD_ENABLE)) != 0) {
2169     mutex_exit(&rdc_conf_lock);
2170     rdc_dev_close(krdc);
2171     if (rc == RDC_EQNOADD) {
2172         spcs_s_add(kstatus, RDC_EQNOADD, rdc_set->disk_queue);
2173         return (RDC_EQNOADD);

```

```

2174     } else {
2175         spcs_s_add(kstatus, RDC_EGROUP,
2176                 rdc_set->primary.intf, rdc_set->primary.file,
2177                 rdc_set->secondary.intf, rdc_set->secondary.file,
2178                 rdc_set->group_name);
2179         return (RDC_EGROUP);
2180     }
2181 }

2183 /*
2184  * maxfbas was set in rdc_dev_open as primary's maxfbas.
2185  * If diskq's maxfbas is smaller, then use diskq's.
2186  */
2187 grp = krdc->group;
2188 if (grp && RDC_IS_DISKQ(grp) && (grp->diskqfd != 0)) {
2189     rc = _rdc_rsrv_diskq(grp);
2190     if (RDC_SUCCESS(rc)) {
2191         rc = nsc_maxfbas(grp->diskqfd, 0, &maxfbas);
2192         if (rc == 0) {
2193             #ifdef DEBUG
2194                 if (krdc->maxfbas != maxfbas)
2195                     cmn_err(CE_NOTE,
2196                             "!_rdc_enable: diskq maxfbas = %"
2197                             NSC_SZFMT ", primary maxfbas = %"
2198                             NSC_SZFMT, maxfbas, krdc->maxfbas);
2199             #endif
2200             krdc->maxfbas = min(krdc->maxfbas, maxfbas);
2201         } else {
2202             cmn_err(CE_WARN,
2203                     "!_rdc_enable: diskq maxfbas failed (%d)",
2204                     rc);
2205         }
2206         _rdc_rlse_diskq(grp);
2207     } else {
2208         cmn_err(CE_WARN,
2209                 "!_rdc_enable: diskq reserve failed (%d)", rc);
2210     }
2211 }

2213 rdc_init_flags(urdc);
2214 (void) strncpy(urdc->direct_file, rdc_set->direct_file, NSC_MAXPATH);
2215 if ((options & RDC_OPT_PRIMARY) && rdc_set->direct_file[0]) {
2216     if (rdc_open_direct(krdc) == NULL)
2217         rdc_set_flags(urdc, RDC_FCAL_FAILED);
2218 }

2220 krdc->many_next = krdc;

2222 ASSERT(krdc->type_flag == 0);
2223 krdc->type_flag = RDC_CONFIGURED;

2225 if (options & RDC_OPT_PRIMARY)
2226     rdc_set_flags(urdc, RDC_PRIMARY);

2228 if (options & RDC_OPT_ASYNC)
2229     krdc->type_flag |= RDC_ASYNCMODE;

2231 set_busy(krdc);
2232 urdc->syshostid = rdc_set->syshostid;

2234 if (add_to_many(krdc) < 0) {
2235     mutex_exit(&rdc_conf_lock);
2237     rdc_group_enter(krdc);
2239     spcs_s_add(kstatus, RDC_EMULTI);

```

```

2240         rc = RDC_EMULTI;
2241         goto fail;
2242     }

2244     /* Configured but not enabled */
2245     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

2247     mutex_exit(&rdc_conf_lock);

2249     rdc_group_enter(krdc);

2251     /* Configured but not enabled */
2252     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

2254     /*
2255      * The rdc set is configured but not yet enabled. Other operations must
2256      * ignore this set until it is enabled.
2257      */

2259     urdc->sync_pos = 0;

2261     if (rdc_set->maxqfbas > 0)
2262         urdc->maxqfbas = rdc_set->maxqfbas;
2263     else
2264         urdc->maxqfbas = rdc_maxthres_queue;

2266     if (rdc_set->maxqitems > 0)
2267         urdc->maxqitems = rdc_set->maxqitems;
2268     else
2269         urdc->maxqitems = rdc_max_qitems;

2271     if (rdc_set->asyncthr > 0)
2272         urdc->asyncthr = rdc_set->asyncthr;
2273     else
2274         urdc->asyncthr = rdc_asyncthr;

2276     if (urdc->autosync == -1) {
2277         /* Still unknown */
2278         if (rdc_set->autosync > 0)
2279             urdc->autosync = 1;
2280         else
2281             urdc->autosync = 0;
2282     }

2284     urdc->netconfig = rdc_set->netconfig;

2286     if (options & RDC_OPT_PRIMARY) {
2287         rhost = rdc_set->secondary.intf;
2288         addrp = &rdc_set->secondary.addr;
2289     } else {
2290         rhost = rdc_set->primary.intf;
2291         addrp = &rdc_set->primary.addr;
2292     }

2294     if (options & RDC_OPT_ASYNC)
2295         rdc_set_flags(urdc, RDC_ASYNC);

2297     svp = rdc_create_svinfo(rhost, addrp, urdc->netconfig);
2298     if (svp == NULL) {
2299         spcs_s_add(kstatus, ENOMEM);
2300         rc = ENOMEM;
2301         goto fail;
2302     }
2303     urdc->netconfig = NULL;          /* This will be no good soon */

2305     rdc_kstat_create(index);

```

```

2307     /* Don't set krdc->intf here */

2309     if (rdc_enable_bitmap(krdc, options & RDC_OPT_SETBMP) < 0)
2310         goto bmpfail;

2312     RDC_ZERO_BITREF(krdc);
2313     if (krdc->lsrv == NULL)
2314         krdc->lsrv = svp;
2315     else {
2316 #ifdef DEBUG
2317         cmn_err(CE_WARN, "!rdc_enable: krdc->lsrv already set: %p",
2318             (void *) krdc->lsrv);
2319 #endif
2320         rdc_destroy_svinfo(svp);
2321     }
2322     svp = NULL;

2324     /* Configured but not enabled */
2325     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

2327     /* And finally */

2329     krdc->remote_index = -1;
2330     /* Should we set the whole group logging? */
2331     rdc_set_flags(urdc, RDC_ENABLED | RDC_LOGGING);

2333     rdc_group_exit(krdc);

2335     if (rdc_intercept(krdc) != 0) {
2336         rdc_group_enter(krdc);
2337         rdc_clr_flags(urdc, RDC_ENABLED);
2338         if (options & RDC_OPT_PRIMARY)
2339             spcs_s_add(kstatus, RDC_EREREGISTER, urdc->primary.file);
2340         else
2341             spcs_s_add(kstatus, RDC_EREREGISTER,
2342                 urdc->secondary.file);
2343 #ifdef DEBUG
2344         cmn_err(CE_NOTE, "Insc_register_path failed %s",
2345             urdc->primary.file);
2346 #endif
2347         rc = RDC_EREREGISTER;
2348         goto bmpfail;
2349     }
2350 #ifdef DEBUG
2351     cmn_err(CE_NOTE, "!SNDR: enabled %s %s", urdc->primary.file,
2352         urdc->secondary.file);
2353 #endif

2355     rdc_write_state(urdc);

2357     mutex_enter(&rdc_conf_lock);
2358     wakeup_busy(krdc);
2359     mutex_exit(&rdc_conf_lock);

2361     return (0);

2363 bmpfail:
2364     if (options & RDC_OPT_PRIMARY)
2365         spcs_s_add(kstatus, RDC_EBITMAP, rdc_set->primary.bitmap);
2366     else
2367         spcs_s_add(kstatus, RDC_EBITMAP, rdc_set->secondary.bitmap);
2368     rc = RDC_EBITMAP;
2369     if (rdc_get_vflags(urdc) & RDC_ENABLED) {
2370         rdc_group_exit(krdc);
2371         (void) rdc_unintercept(krdc);

```

```

2372         rdc_group_enter(krdc);
2373     }
2375 fail:
2376     rdc_kstat_delete(index);
2377     rdc_group_exit(krdc);
2378     if (krdc->intf) {
2379         rdc_if_t *ip = krdc->intf;
2380         mutex_enter(&rdc_conf_lock);
2381         krdc->intf = NULL;
2382         rdc_remove_from_if(ip);
2383         mutex_exit(&rdc_conf_lock);
2384     }
2385     rdc_group_enter(krdc);
2386     /* Configured but not enabled */
2387     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));
2389     rdc_dev_close(krdc);
2390     rdc_close_direct(krdc);
2391     rdc_destroy_svinfo(svp);
2393     /* Configured but not enabled */
2394     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));
2396     rdc_group_exit(krdc);
2398     mutex_enter(&rdc_conf_lock);
2400     /* Configured but not enabled */
2401     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));
2403     remove_from_group(krdc);
2405     if (IS_MANY(krdc) || IS_MULTI(krdc))
2406         remove_from_many(krdc);
2408     rdc_u_init(urdc);
2410     ASSERT(krdc->type_flag & RDC_CONFIGURED);
2411     krdc->type_flag = 0;
2412     wakeup_busy(krdc);
2414     mutex_exit(&rdc_conf_lock);
2416     return (rc);
2417 }
2419 static int
2420 rdc_enable(rdc_config_t *uparms, spcs_s_info_t kstatus)
2421 {
2422     int rc;
2423     char itmp[10];
2425     if (!(uparms->options & RDC_OPT_SYNC) &&
2426         !(uparms->options & RDC_OPT_ASYNC)) {
2427         rc = RDC_EEINVAL;
2428         (void) spcs_s_inttostring(
2429             uparms->options, itmp, sizeof(itmp), 1);
2430         spcs_s_add(kstatus, RDC_EEINVAL, itmp);
2431         goto done;
2432     }
2434     if (!(uparms->options & RDC_OPT_PRIMARY) &&
2435         !(uparms->options & RDC_OPT_SECONDARY)) {
2436         rc = RDC_EEINVAL;
2437         (void) spcs_s_inttostring(

```

```

2438         uparms->options, itmp, sizeof(itmp), 1);
2439         spcs_s_add(kstatus, RDC_EEINVAL, itmp);
2440         goto done;
2441     }
2443     if (!(uparms->options & RDC_OPT_SETBMP) &&
2444         !(uparms->options & RDC_OPT_CLRBMP)) {
2445         rc = RDC_EEINVAL;
2446         (void) spcs_s_inttostring(
2447             uparms->options, itmp, sizeof(itmp), 1);
2448         spcs_s_add(kstatus, RDC_EEINVAL, itmp);
2449         goto done;
2450     }
2452     rc = _rdc_enable(uparms->rdc_set, uparms->options, kstatus);
2453 done:
2454     return (rc);
2455 }
2457 /* ARGSUSED */
2458 static int
2459 _rdc_disable(rdc_k_info_t *krdc, rdc_config_t *uap, spcs_s_info_t kstatus)
2460 {
2461     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
2462     rdc_if_t *ip;
2463     int index = krdc->index;
2464     disk_queue *q;
2465     rdc_set_t *rdc_set = uap->rdc_set;
2467     ASSERT(krdc->group != NULL);
2468     rdc_group_enter(krdc);
2469 #ifdef DEBUG
2470     ASSERT(rdc_check(krdc, rdc_set) == 0);
2471 #else
2472     if (((uap->options & RDC_OPT_FORCE_DISABLE) == 0) &&
2473         rdc_check(krdc, rdc_set)) {
2474         rdc_group_exit(krdc);
2475         spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
2476                 rdc_set->secondary.file);
2477         return (RDC_EALREADY);
2478     }
2479 #endif
2481     if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
2482         halt_sync(krdc);
2483         ASSERT(IS_ENABLED(urdc));
2484     }
2485     q = &krdc->group->diskq;
2487     if (IS_ASYNC(urdc) && RDC_IS_DISKQ(krdc->group) &&
2488         ((!IS_STATE(urdc, RDC_LOGGING)) && (!QEMPTY(q)))) {
2489         krdc->type_flag &= ~RDC_DISABLEPEND;
2490         rdc_group_exit(krdc);
2491         spcs_s_add(kstatus, RDC_EQNOTEMPTY, urdc->disk_queue);
2492         return (RDC_EQNOTEMPTY);
2493     }
2494     rdc_group_exit(krdc);
2495     (void) rdc_unintercept(krdc);
2497 #ifdef DEBUG
2498     cmn_err(CE_NOTE, "!SNDR: disabled %s %s", urdc->primary.file,
2499            urdc->secondary.file);
2500 #endif
2502     /* Configured but not enabled */
2503     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

```

```

2505     /*
2506     * No new io can come in through the io provider.
2507     * Wait for the async flusher to finish.
2508     */

2510     if (IS_ASYNC(urdc) && !RDC_IS_DISKQ(krdc->group)) {
2511         int tries = 2; /* in case of hopelessly stuck flusher threads */
2512 #ifdef DEBUG
2513         net_queue *qp = &krdc->group->ra_queue;
2514 #endif
2515         do {
2516             if (!krdc->group->rdc_writer)
2517                 (void) rdc_writer(krdc->index);

2519             (void) rdc_drain_queue(krdc->index);

2521         } while (krdc->group->rdc_writer && tries--);

2523         /* ok, force it to happen... */
2524         if (rdc_drain_queue(krdc->index) != 0) {
2525             do {
2526                 mutex_enter(&krdc->group->ra_queue.net_qlock);
2527                 krdc->group->asynctdis = 1;
2528                 cv_broadcast(&krdc->group->asynctcv);
2529                 mutex_exit(&krdc->group->ra_queue.net_qlock);
2530                 cmn_err(CE_WARN,
2531                      "!SNDR: async I/O pending and not flushed "
2532                      "for %s during disable",
2533                      urdc->primary.file);
2534 #ifdef DEBUG
2535                 cmn_err(CE_WARN,
2536                      "Initems: %" NSC_SZFMT " nblocks: %"
2537                      NSC_SZFMT " head: 0x%p tail: 0x%p",
2538                      qp->nitems, qp->nblocks,
2539                      (void *)qp->net_qhead,
2540                      (void *)qp->net_qtail);
2541 #endif
2542             } while (krdc->group->rdc_thrnum > 0);
2543         }
2544     }

2546     mutex_enter(&rdc_conf_lock);
2547     ip = krdc->intf;
2548     krdc->intf = 0;

2550     if (ip) {
2551         rdc_remove_from_if(ip);
2552     }

2554     mutex_exit(&rdc_conf_lock);

2556     rdc_group_enter(krdc);

2558     /* Configured but not enabled */
2559     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

2561     /* Must not hold group lock during this function */
2562     rdc_group_exit(krdc);
2563     while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
2564         delay(2);
2565     rdc_group_enter(krdc);

2567     (void) rdc_clear_state(krdc);

2569     rdc_free_bitmap(krdc, RDC_CMD_DISABLE);

```

```

2570     rdc_close_bitmap(krdc);

2572     rdc_dev_close(krdc);
2573     rdc_close_direct(krdc);

2575     /* Configured but not enabled */
2576     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

2578     rdc_group_exit(krdc);

2580     /*
2581     * we should now unregister the queue, with no conflicting
2582     * locks held. This is the last(only) member of the group
2583     */
2584     if (krdc->group && RDC_IS_DISKQ(krdc->group) &&
2585         krdc->group->count == 1) { /* stop protecting queue */
2586         rdc_unintercept_diskq(krdc->group);
2587     }

2589     mutex_enter(&rdc_conf_lock);

2591     /* Configured but not enabled */
2592     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

2594     wait_busy(krdc);

2596     if (IS_MANY(krdc) || IS_MULTII(krdc))
2597         remove_from_many(krdc);

2599     remove_from_group(krdc);

2601     krdc->remote_index = -1;
2602     ASSERT(krdc->type_flag & RDC_CONFIGURED);
2603     ASSERT(krdc->type_flag & RDC_DISABLEPEND);
2604     krdc->type_flag = 0;
2605 #ifdef DEBUG
2606     if (krdc->dcio_bitmap)
2607         cmn_err(CE_WARN, "!_rdc_disable: possible mem leak, "
2608              "dcio_bitmap");
2609 #endif
2610     krdc->dcio_bitmap = NULL;
2611     krdc->bitmap_ref = NULL;
2612     krdc->bitmap_size = 0;
2613     krdc->maxfbas = 0;
2614     krdc->bitmap_write = 0;
2615     krdc->disk_status = 0;
2616     rdc_destroy_svinfo(krdc->lsrv);
2617     krdc->lsrv = NULL;
2618     krdc->multi_next = NULL;

2620     rdc_u_init(urdc);

2622     mutex_exit(&rdc_conf_lock);
2623     rdc_kstat_delete(index);

2625     return (0);
2626 }

2628 static int
2629 rdc_disable(rdc_config_t *uparms, spcs_s_info_t kstatus)
2630 {
2631     rdc_k_info_t *krdc;
2632     int index;
2633     int rc;

2635     mutex_enter(&rdc_conf_lock);

```

```

2637     index = rdc_lookup_byname(uparms->rdc_set);
2638     if (index >= 0)
2639         krdc = &rdc_k_info[index];
2640     if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
2641         mutex_exit(&rdc_conf_lock);
2642         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
2643             uparms->rdc_set->secondary.file);
2644         return (RDC_EALREADY);
2645     }
2647     krdc->type_flag |= RDC_DISABLEPEND;
2648     wait_busy(krdc);
2649     if (krdc->type_flag == 0) {
2650         /* A resume or enable failed */
2651         mutex_exit(&rdc_conf_lock);
2652         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
2653             uparms->rdc_set->secondary.file);
2654         return (RDC_EALREADY);
2655     }
2656     mutex_exit(&rdc_conf_lock);
2658     rc = _rdc_disable(krdc, uparms, kstatus);
2659     return (rc);
2660 }
2663 /*
2664 * Checks whether the state of one of the other sets in the 1-many or
2665 * multi-hop config should prevent a sync from starting on this one.
2666 * Return NULL if no just cause or impediment is found, otherwise return
2667 * a pointer to the offending set.
2668 */
2669 static rdc_u_info_t *
2670 rdc_allow_pri_sync(rdc_u_info_t *urdc, int options)
2671 {
2672     rdc_k_info_t *krdc = &rdc_k_info[urdc->index];
2673     rdc_k_info_t *ktmp;
2674     rdc_u_info_t *utmp;
2675     rdc_k_info_t *kmulti = NULL;
2677     ASSERT(rdc_get_vflags(urdc) & RDC_PRIMARY);
2679     rdc_many_enter(krdc);
2681     /*
2682     * In the reverse sync case we need to check the previous leg of
2683     * the multi-hop config. The link to that set can be from any of
2684     * the 1-many list, so as we go through we keep an eye open for it.
2685     */
2686     if ((options & RDC_OPT_REVERSE) && (IS_MULTI(krdc))) {
2687         /* This set links to the first leg */
2688         ktmp = krdc->multi_next;
2689         utmp = &rdc_u_info[ktmp->index];
2690         if (IS_ENABLED(utmp))
2691             kmulti = ktmp;
2692     }
2694     if (IS_MANY(krdc)) {
2695         for (ktmp = krdc->many_next; ktmp != krdc;
2696             ktmp = ktmp->many_next) {
2697             utmp = &rdc_u_info[ktmp->index];
2699             if (!IS_ENABLED(utmp))
2700                 continue;

```

```

2702         if (options & RDC_OPT_FORWARD) {
2703             /*
2704             * Reverse sync needed is bad, as it means a
2705             * reverse sync in progress or started and
2706             * didn't complete, so this primary volume
2707             * is not consistent. So we shouldn't copy
2708             * it to its secondary.
2709             */
2710             if (rdc_get_mflags(utmp) & RDC_RSYNC_NEEDED) {
2711                 rdc_many_exit(krdc);
2712                 return (utmp);
2713             }
2714         } else {
2715             /* Reverse, so see if we need to spot kmulti */
2716             if ((kmulti == NULL) && (IS_MULTI(ktmp))) {
2717                 /* This set links to the first leg */
2718                 kmulti = ktmp->multi_next;
2719                 if (!IS_ENABLED(
2720                     &rdc_u_info[kmulti->index]))
2721                     kmulti = NULL;
2722             }
2724             /*
2725             * Non-logging is bad, as the bitmap will
2726             * be updated with the bits for this sync.
2727             */
2728             if (!(rdc_get_vflags(utmp) & RDC_LOGGING)) {
2729                 rdc_many_exit(krdc);
2730                 return (utmp);
2731             }
2732         }
2733     }
2734 }
2736 if (kmulti) {
2737     utmp = &rdc_u_info[kmulti->index];
2738     ktmp = kmulti; /* In case we decide we do need to use ktmp */
2740     ASSERT(options & RDC_OPT_REVERSE);
2742     if (IS_REPLICATING(utmp)) {
2743         /*
2744         * Replicating is bad as data is already flowing to
2745         * the target of the requested sync operation.
2746         */
2747         rdc_many_exit(krdc);
2748         return (utmp);
2749     }
2751     if (rdc_get_vflags(utmp) & RDC_SYNCING) {
2752         /*
2753         * Forward sync in progress is bad, as data is
2754         * already flowing to the target of the requested
2755         * sync operation.
2756         * Reverse sync in progress is bad, as the primary
2757         * has already decided which data to copy.
2758         */
2759         rdc_many_exit(krdc);
2760         return (utmp);
2761     }
2763     /*
2764     * Clear the "sync needed" flags, as the multi-hop secondary
2765     * will be updated via this requested sync operation, so does
2766     * not need to complete its aborted forward sync.
2767     */

```



```

2768         if (rdc_get_vflags(utmp) & RDC_SYNC_NEEDED)
2769             rdc_clr_flags(utmp, RDC_SYNC_NEEDED);
2770     }

2772     if (IS_MANY(krdc) && (options & RDC_OPT_REVERSE)) {
2773         for (ktmp = krdc->many_next; ktmp != krdc;
2774             ktmp = ktmp->many_next) {
2775             utmp = &rdc_u_info[ktmp->index];
2776             if (!IS_ENABLED(utmp))
2777                 continue;

2779             /*
2780              * Clear any "reverse sync needed" flags, as the
2781              * volume will be updated via this requested
2782              * sync operation, so does not need to complete
2783              * its aborted reverse sync.
2784              */
2785             if (rdc_get_mflags(utmp) & RDC_RSsync_NEEDED)
2786                 rdc_clr_mflags(utmp, RDC_RSsync_NEEDED);
2787         }
2788     }

2790     rdc_many_exit(krdc);

2792     return (NULL);
2793 }

2795 static void
2796 _rdc_sync_wrthr(void *thrinfo)
2797 {
2798     rdc_syncthr_t *syncinfo = (rdc_syncthr_t *)thrinfo;
2799     nsc_buf_t *handle = NULL;
2800     rdc_k_info_t *krdc = syncinfo->krdc;
2801     int rc;
2802     int tries = 0;

2804     DTRACE_PROBE2(rdc_sync_loop_netwrite_start, int, krdc->index,
2805                  nsc_buf_t *, handle);

2807 retry:
2808     rc = nsc_alloc_buf(RDC_U_FD(krdc), syncinfo->offset, syncinfo->len,
2809                      NSC_READ | NSC_NOCACHE, &handle);

2811     if (!RDC_SUCCESS(rc) || krdc->remote_index < 0) {
2812         DTRACE_PROBE(rdc_sync_wrthr_alloc_buf_err);
2813         goto failed;
2814     }

2816     rdc_group_enter(krdc);
2817     if ((krdc->disk_status == 1) || (krdc->dcio_bitmap == NULL)) {
2818         rdc_group_exit(krdc);
2819         goto failed;
2820     }
2821     rdc_group_exit(krdc);

2823     if ((rc = rdc_net_write(krdc->index, krdc->remote_index, handle,
2824                          handle->sb_pos, handle->sb_len, RDC_NOSEQ, RDC_NOQUE, NULL)) > 0) {
2825         rdc_u_info_t *urdc = &rdc_u_info[krdc->index];

2827         /*
2828          * The following is to handle
2829          * the case where the secondary side
2830          * has thrown our buffer handle token away in a
2831          * attempt to preserve its health on restart
2832          */
2833         if ((rc == EPROTO) && (tries < 3)) {

```

```

2834             (void) nsc_free_buf(handle);
2835             handle = NULL;
2836             tries++;
2837             delay(HZ >> 2);
2838             goto retry;
2839         }

2841         DTRACE_PROBE(rdc_sync_wrthr_remote_write_err);
2842         cmn_err(CE_WARN, "!rdc_sync_wrthr: remote write failed (%d) "
2843              "0x%x", rc, rdc_get_vflags(urdc));

2845         goto failed;
2846     }
2847     (void) nsc_free_buf(handle);
2848     handle = NULL;

2850     return;
2851 failed:
2852     (void) nsc_free_buf(handle);
2853     syncinfo->status->offset = syncinfo->offset;
2854 }

2856 /*
2857  * see above comments on _rdc_sync_wrthr
2858  */
2859 static void
2860 _rdc_sync_rdthr(void *thrinfo)
2861 {
2862     rdc_syncthr_t *syncinfo = (rdc_syncthr_t *)thrinfo;
2863     nsc_buf_t *handle = NULL;
2864     rdc_k_info_t *krdc = syncinfo->krdc;
2865     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
2866     int rc;

2868     rc = nsc_alloc_buf(RDC_U_FD(krdc), syncinfo->offset, syncinfo->len,
2869                      NSC_WRITE | NSC_WRTHRU | NSC_NOCACHE, &handle);

2871     if (!RDC_SUCCESS(rc) || krdc->remote_index < 0) {
2872         goto failed;
2873     }

2874     rdc_group_enter(krdc);
2875     if ((krdc->disk_status == 1) || (krdc->dcio_bitmap == NULL)) {
2876         rdc_group_exit(krdc);
2877         goto failed;
2878     }
2879     rdc_group_exit(krdc);

2881     rc = rdc_net_read(krdc->index, krdc->remote_index, handle,
2882                    handle->sb_pos, handle->sb_len);

2884     if (!RDC_SUCCESS(rc)) {
2885         cmn_err(CE_WARN, "!rdc_sync_rdthr: remote read failed(%d)", rc);
2886         goto failed;
2887     }
2888     if (!IS_STATE(urdc, RDC_FULL))
2889         rdc_set_bitmap_many(krdc, handle->sb_pos, handle->sb_len);

2891     rc = nsc_write(handle, handle->sb_pos, handle->sb_len, 0);

2893     if (!RDC_SUCCESS(rc)) {
2894         rdc_many_enter(krdc);
2895         rdc_set_flags_log(urdc, RDC_VOL_FAILED, "nsc_write failed");
2896         rdc_many_exit(krdc);
2897         rdc_write_state(urdc);
2898         goto failed;
2899     }

```

```

2901     (void) nsc_free_buf(handle);
2902     handle = NULL;

2904     return;
2905 failed:
2906     (void) nsc_free_buf(handle);
2907     syncinfo->status->offset = syncinfo->offset;
2908 }

2910 /*
2911  * _rdc_sync_wrthr
2912  * sync loop write thread
2913  * if there are avail threads, we have not
2914  * used up the pipe, so the sync loop will, if
2915  * possible use these to multithread the write/read
2916  */
2917 void
2918 _rdc_sync_thread(void *thrinfo)
2919 {
2920     rdc_syncthr_t *syncinfo = (rdc_syncthr_t *)thrinfo;
2921     rdc_k_info_t *krdc = syncinfo->krdc;
2922     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
2923     rdc_thrsync_t *sync = &krdc->syncs;
2924     uint_t bitmask;
2925     int rc;

2927     rc = _rdc_rsrv_devs(krdc, RDC_RAW, RDC_INTERNAL);
2928     if (!RDC_SUCCESS(rc))
2929         goto failed;

2931     if (IS_STATE(urdc, RDC_SLAVE))
2932         _rdc_sync_rdthr(thrinfo);
2933     else
2934         _rdc_sync_wrthr(thrinfo);

2936     _rdc_rlse_devs(krdc, RDC_RAW);

2938     if (krdc->dcio_bitmap == NULL) {
2939 #ifdef DEBUG
2940         cmn_err(CE_NOTE, "!_rdc_sync_wrthr: NULL bitmap");
2941 #else
2942         /*EMPTY*/
2943 #endif
2944     } else if (syncinfo->status->offset < 0) {

2946         RDC_SET_BITMASK(syncinfo->offset, syncinfo->len, &bitmask);
2947         RDC_CLR_BITMAP(krdc, syncinfo->offset, syncinfo->len, \
2948             bitmask, RDC_BIT_FORCE);
2949     }

2951 failed:
2952     /*
2953      * done with this, get rid of it.
2954      * the status is not freed, it should still be a status chain
2955      * that _rdc_sync() has the head of
2956      */
2957     kmem_free(syncinfo, sizeof (*syncinfo));

2959     /*
2960      * decrement the global sync thread num
2961      */
2962     mutex_enter(&sync_info.lock);
2963     sync_info.active_thr--;
2964     /* LINTED */
2965     RDC_AVAIL_THR_TUNE(sync_info);

```

```

2966     mutex_exit(&sync_info.lock);

2968     /*
2969      * krdc specific stuff
2970      */
2971     mutex_enter(&sync->lock);
2972     sync->complete++;
2973     cv_broadcast(&sync->cv);
2974     mutex_exit(&sync->lock);
2975 }

2977 int
2978 _rdc_setup_syncthr(rdc_syncthr_t **synthr, nsc_off_t offset,
2979     nsc_size_t len, rdc_k_info_t *krdc, sync_status_t *stats)
2980 {
2981     rdc_syncthr_t *tmp;
2982     /* alloc here, free in the sync thread */
2983     tmp =
2984         (rdc_syncthr_t *)kmem_zalloc(sizeof (rdc_syncthr_t), KM_NOSLEEP);

2986     if (tmp == NULL)
2987         return (-1);
2988     tmp->offset = offset;
2989     tmp->len = len;
2990     tmp->status = stats;
2991     tmp->krdc = krdc;

2993     *synthr = tmp;
2994     return (0);
2995 }

2997 sync_status_t *
2998 _rdc_new_sync_status()
2999 {
3000     sync_status_t *s;

3002     s = (sync_status_t *)kmem_zalloc(sizeof (*s), KM_NOSLEEP);
3003     s->offset = -1;
3004     return (s);
3005 }

3007 void
3008 _rdc_free_sync_status(sync_status_t *status)
3009 {
3010     sync_status_t *s;

3012     while (status) {
3013         s = status->next;
3014         kmem_free(status, sizeof (*status));
3015         status = s;
3016     }
3017 }

3018 int
3019 _rdc_sync_status_ok(sync_status_t *status, int *offset)
3020 {
3021 #ifdef DEBUG_SYNCSTATUS
3022     int i = 0;
3023 #endif
3024     while (status) {
3025         if (status->offset >= 0) {
3026             *offset = status->offset;
3027             return (-1);
3028         }
3029         status = status->next;
3030 #ifdef DEBUG_SYNCSTATUS
3031         i++;

```

```

3032 #endif
3033     }
3034 #ifdef DEBUGSYNCSTATUS
3035     cmn_err(CE_NOTE, "rdc_sync_status_ok: checked %d statuses", i);
3036 #endif
3037     return (0);
3038 }

3040 int mtsync = 1;
3041 /*
3042  * _rdc_sync() : rdc sync loop
3043  */
3044 */
3045 static void
3046 _rdc_sync(rdc_k_info_t *krdc)
3047 {
3048     nsc_size_t size = 0;
3049     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
3050     int rtype;
3051     int sts;
3052     int reserved = 0;
3053     nsc_buf_t *alloc_h = NULL;
3054     nsc_buf_t *handle = NULL;
3055     nsc_off_t mask;
3056     nsc_size_t maxbit;
3057     nsc_size_t len;
3058     nsc_off_t offset = 0;
3059     int sync_completed = 0;
3060     int tries = 0;
3061     int rc;
3062     int queuing = 0;
3063     uint_t bitmask;
3064     sync_status_t *ss, *sync_status = NULL;
3065     rdc_thrsync_t *sync = &krdc->syncs;
3066     rdc_syncthr_t *syncinfo;
3067     nsthread_t *trc = NULL;

3069     if (IS_STATE(urdc, RDC_QUEUEING) && !IS_STATE(urdc, RDC_FULL)) {
3070         /* flusher is handling the sync in the update case */
3071         queuing = 1;
3072         goto sync_done;
3073     }

3075     /*
3076      * Main sync/resync loop
3077      */
3078     DTRACE_PROBE(rdc_sync_loop_start);

3080     rtype = RDC_RAW;
3081     sts = _rdc_rsrv_devs(krdc, rtype, RDC_INTERNAL);

3083     DTRACE_PROBE(rdc_sync_loop_rsrv);

3085     if (sts != 0)
3086         goto failed_noincr;

3088     reserved = 1;

3090     /*
3091      * pre-allocate a handle if we can - speeds up the sync.
3092      */

3094     if (rdc_prealloc_handle) {
3095         alloc_h = nsc_alloc_handle(RDC_U_FD(krdc), NULL, NULL, NULL);
3096 #ifdef DEBUG
3097         if (!alloc_h) {

```

```

3098         cmn_err(CE_WARN,
3099                 "rdc_sync: failed to pre-alloc handle");
3100     }
3101 #endif
3102     } else {
3103         alloc_h = NULL;
3104     }

3106     ASSERT(urdc->volume_size != 0);
3107     size = urdc->volume_size;
3108     mask = ~(LOG_TO_FBA_NUM(1) - 1);
3109     maxbit = FBA_TO_LOG_NUM(size - 1);

3111     /*
3112      * as this while loop can also move data, it is counted as a
3113      * sync loop thread
3114      */
3115     rdc_group_enter(krdc);
3116     rdc_clr_flags(urdc, RDC_LOGGING);
3117     rdc_set_flags(urdc, RDC_SYNCING);
3118     krdc->group->synccount++;
3119     rdc_group_exit(krdc);
3120     mutex_enter(&sync_info.lock);
3121     sync_info.active_thr++;
3122     /* LINTED */
3123     RDC_AVAIL_THR_TUNE(sync_info);
3124     mutex_exit(&sync_info.lock);

3126     while (offset < size) {
3127         rdc_group_enter(krdc);
3128         ASSERT(krdc->aux_state & RDC_AUXSYNCIP);
3129         if (krdc->disk_status == 1 || krdc->dcio_bitmap == NULL) {
3130             rdc_group_exit(krdc);
3131             if (krdc->disk_status == 1) {
3132                 DTRACE_PROBE(rdc_sync_loop_disk_status_err);
3133             } else {
3134                 DTRACE_PROBE(rdc_sync_loop_dcio_bitmap_err);
3135             }
3136             goto failed; /* halt sync */
3137         }
3138         rdc_group_exit(krdc);

3140         if (!(rdc_get_vflags(urdc) & RDC_FULL)) {
3141             mutex_enter(&krdc->syncbitmutex);
3142             krdc->syncbitpos = FBA_TO_LOG_NUM(offset);
3143             len = 0;

3145             /* skip unnecessary chunks */

3147             while (krdc->syncbitpos <= maxbit &&
3148                    !RDC_BIT_ISSET(krdc, krdc->syncbitpos)) {
3149                 offset += LOG_TO_FBA_NUM(1);
3150                 krdc->syncbitpos++;
3151             }

3153             /* check for boundary */

3155             if (offset >= size) {
3156                 mutex_exit(&krdc->syncbitmutex);
3157                 goto sync_done;
3158             }

3160             /* find maximal length we can transfer */

3162             while (krdc->syncbitpos <= maxbit &&
3163                    RDC_BIT_ISSET(krdc, krdc->syncbitpos)) {

```

```

3164         len += LOG_TO_FBA_NUM(1);
3165         krdc->syncbitpos++;
3166         /* we can only read maxfbas anyways */
3167         if (len >= krdc->maxfbas)
3168             break;
3169     }
3171     len = min(len, (size - offset));
3173 } else {
3174     len = size - offset;
3175 }
3177 /* truncate to the io provider limit */
3178 ASSERT(krdc->maxfbas != 0);
3179 len = min(len, krdc->maxfbas);
3181 if (len > LOG_TO_FBA_NUM(1)) {
3182     /*
3183      * If the update is larger than a bitmap chunk,
3184      * then truncate to a whole number of bitmap
3185      * chunks.
3186      *
3187      * If the update is smaller than a bitmap
3188      * chunk, this must be the last write.
3189      */
3190     len &= mask;
3191 }
3193 if (!(rdc_get_vflags(urdc) & RDC_FULL)) {
3194     krdc->syncbitpos = FBA_TO_LOG_NUM(offset + len);
3195     mutex_exit(&krdc->syncbitmutex);
3196 }
3198 /*
3199  * Find out if we can reserve a thread here ...
3200  * note: skip the mutex for the first check, if the number
3201  * is up there, why bother even grabbing the mutex to
3202  * only realize that we can't have a thread anyways
3203  */
3205 if (mtsync && sync_info.active_thr < RDC_MAX_SYNC_THREADS) {
3207     mutex_enter(&sync_info.lock);
3208     if (sync_info.avail_thr >= 1) {
3209         if (sync_status == NULL) {
3210             ss = sync_status =
3211                 _rdc_new_sync_status();
3212         } else {
3213             ss = ss->next = _rdc_new_sync_status();
3214         }
3215         if (ss == NULL) {
3216             mutex_exit(&sync_info.lock);
3217             #ifdef DEBUG
3218                 cmn_err(CE_WARN, "!rdc_sync: can't "
3219                     "allocate status for mt sync");
3220             #endif
3221             goto retry;
3222         }
3223     }
3224     /* syncinfo protected by sync_info lock but
3225      * not part of the sync_info structure
3226      * be careful if moving
3227      */
3228     if (_rdc_setup_syncthr(&syncinfo,
3229         offset, len, krdc, ss) < 0) {

```

```

3230         _rdc_free_sync_status(ss);
3231     }
3233     trc = nst_create(sync_info.rdc_syncset,
3234         _rdc_sync_thread, syncinfo, NST_SLEEP);
3236     if (trc == NULL) {
3237         mutex_exit(&sync_info.lock);
3238         #ifdef DEBUG
3239             cmn_err(CE_NOTE, "!rdc_sync: unable to "
3240                 "mt sync");
3241         #endif
3242         _rdc_free_sync_status(ss);
3243         kmem_free(syncinfo, sizeof (*syncinfo));
3244         syncinfo = NULL;
3245         goto retry;
3246     } else {
3247         mutex_enter(&sync->lock);
3248         sync->threads++;
3249         mutex_exit(&sync->lock);
3250     }
3252     sync_info.active_thr++;
3253     /* LINTED */
3254     RDC_AVAIL_THR_TUNE(sync_info);
3256     mutex_exit(&sync_info.lock);
3257     goto threaded;
3258 }
3259     mutex_exit(&sync_info.lock);
3260 }
3261 retry:
3262     handle = alloc_h;
3263     DTRACE_PROBE(rdc_sync_loop_allocbuf_start);
3264     if (rdc_get_vflags(urdc) & RDC_SLAVE)
3265         sts = nsc_alloc_buf(RDC_U_FD(krdc), offset, len,
3266             NSC_WRITE | NSC_WRTHRU | NSC_NOCACHE, &handle);
3267     else
3268         sts = nsc_alloc_buf(RDC_U_FD(krdc), offset, len,
3269             NSC_READ | NSC_NOCACHE, &handle);
3271     DTRACE_PROBE(rdc_sync_loop_allocbuf_end);
3272     if (sts > 0) {
3273         if (handle && handle != alloc_h) {
3274             (void) nsc_free_buf(handle);
3275         }
3277         handle = NULL;
3278         DTRACE_PROBE(rdc_sync_loop_allocbuf_err);
3279         goto failed;
3280     }
3282     if (rdc_get_vflags(urdc) & RDC_SLAVE) {
3283         /* overwrite buffer with remote data */
3284         sts = rdc_net_read(krdc->index, krdc->remote_index,
3285             handle, handle->sb_pos, handle->sb_len);
3287         if (!RDC_SUCCESS(sts)) {
3288             #ifdef DEBUG
3289                 cmn_err(CE_WARN,
3290                     "!rdc_sync: remote read failed (%d)", sts);
3291             #endif
3292             DTRACE_PROBE(rdc_sync_loop_remote_read_err);
3293             goto failed;
3294         }
3295         if (!(rdc_get_vflags(urdc) & RDC_FULL))

```

```

3296         rdc_set_bitmap_many(krdc, handle->sb_pos,
3297                             handle->sb_len);
3299     /* commit locally */
3301     sts = nsc_write(handle, handle->sb_pos,
3302                   handle->sb_len, 0);
3304     if (!RDC_SUCCESS(sts)) {
3305         /* reverse sync needed already set */
3306         rdc_many_enter(krdc);
3307         rdc_set_flags_log(urdc, RDC_VOL_FAILED,
3308                         "write failed during sync");
3309         rdc_many_exit(krdc);
3310         rdc_write_state(urdc);
3311         DTRACE_PROBE(rdc_sync_loop_nsc_write_err);
3312         goto failed;
3313     }
3314 } else {
3315     /* send local data to remote */
3316     DTRACE_PROBE2(rdc_sync_loop_netwrite_start,
3317                 int, krdc->index, nsc_buf_t *, handle);
3319     if ((sts = rdc_net_write(krdc->index,
3320                             krdc->remote_index, handle, handle->sb_pos,
3321                             handle->sb_len, RDC_NOSEQ, RDC_NOQUE, NULL)) > 0) {
3323         /*
3324          * The following is to handle
3325          * the case where the secondary side
3326          * has thrown our buffer handle token away in a
3327          * attempt to preserve its health on restart
3328          */
3329         if ((sts == EPROTO) && (tries < 3)) {
3330             (void) nsc_free_buf(handle);
3331             handle = NULL;
3332             tries++;
3333             delay(HZ >> 2);
3334             goto retry;
3335         }
3336 #ifdef DEBUG
3337         cmn_err(CE_WARN,
3338              "lrdc sync: remote write failed (%d) 0x%x",
3339              sts, rdc_get_vflags(urdc));
3340 #endif
3341         DTRACE_PROBE(rdc_sync_loop_netwrite_err);
3342         goto failed;
3343     }
3344     DTRACE_PROBE(rdc_sync_loop_netwrite_end);
3345 }
3347 (void) nsc_free_buf(handle);
3348 handle = NULL;
3350 if (krdc->dcio_bitmap == NULL) {
3351 #ifdef DEBUG
3352     cmn_err(CE_NOTE, "!rdc_sync: NULL bitmap");
3353 #else
3354     ;
3355     /*EMPTY*/
3356 #endif
3357 } else {
3359     RDC_SET_BITMASK(offset, len, &bitmap);
3360     RDC_CLR_BITMAP(krdc, offset, len, bitmap, \
3361                   RDC_BIT_FORCE);

```

```

3362         ASSERT(!IS_ASYNC(urdc));
3363     }
3365     /*
3366      * Only release/reserve if someone is waiting
3367      */
3368     if (krdc->devices->id_release || nsc_waiting(RDC_U_FD(krdc))) {
3369         DTRACE_PROBE(rdc_sync_loop_rlse_start);
3370         if (alloc_h) {
3371             (void) nsc_free_handle(alloc_h);
3372             alloc_h = NULL;
3373         }
3375         _rdc_rlse_devs(krdc, rtype);
3376         reserved = 0;
3377         delay(2);
3379         rtype = RDC_RAW;
3380         sts = _rdc_rsrv_devs(krdc, rtype, RDC_INTERNAL);
3381         if (sts != 0) {
3382             handle = NULL;
3383             DTRACE_PROBE(rdc_sync_loop_rdc_rsrv_err);
3384             goto failed;
3385         }
3387         reserved = 1;
3389         if (rdc_prealloc_handle) {
3390             alloc_h = nsc_alloc_handle(RDC_U_FD(krdc),
3391                                       NULL, NULL, NULL);
3392 #ifdef DEBUG
3393             if (!alloc_h) {
3394                 cmn_err(CE_WARN, "!rdc_sync: "
3395                       "failed to pre-alloc handle");
3396             }
3397 #endif
3398         }
3399         DTRACE_PROBE(rdc_sync_loop_rlse_end);
3400     }
3401 threaded:
3402     offset += len;
3403     urdc->sync_pos = offset;
3404 }
3406 sync_done:
3407     sync_completed = 1;
3409 failed:
3410     krdc->group->synccount--;
3411 failed_noincr:
3412     mutex_enter(&sync->lock);
3413     while (sync->complete != sync->threads) {
3414         cv_wait(&sync->cv, &sync->lock);
3415     }
3416     sync->complete = 0;
3417     sync->threads = 0;
3418     mutex_exit(&sync->lock);
3420     /*
3421      * if sync_completed is 0 here,
3422      * we know that the main sync thread failed anyway
3423      * so just free the statuses and fail
3424      */
3425     if (sync_completed && (_rdc_sync_status_ok(sync_status, &rc) < 0)) {
3426         urdc->sync_pos = rc;
3427         sync_completed = 0; /* at least 1 thread failed */

```

```

3428     }
3430     __rdc_free_sync_status(sync_status);
3432     /*
3433     * we didn't increment, we didn't even sync,
3434     * so don't dec sync_info.active_thr
3435     */
3436     if (!queuing) {
3437         mutex_enter(&sync_info.lock);
3438         sync_info.active_thr--;
3439         /* LINTED */
3440         RDC_AVAIL_THR_TUNE(sync_info);
3441         mutex_exit(&sync_info.lock);
3442     }
3444     if (handle) {
3445         (void) nsc_free_buf(handle);
3446     }
3448     if (alloc_h) {
3449         (void) nsc_free_handle(alloc_h);
3450     }
3452     if (reserved) {
3453         __rdc_rlse_devs(krdc, rtype);
3454     }
3456 notstarted:
3457     rdc_group_enter(krdc);
3458     ASSERT(krdc->aux_state & RDC_AUXSYNCP);
3459     if (IS_STATE(urdc, RDC_QUEUEING))
3460         rdc_clr_flags(urdc, RDC_QUEUEING);
3462     if (sync_completed) {
3463         (void) rdc_net_state(krdc->index, CCIO_DONE);
3464     } else {
3465         (void) rdc_net_state(krdc->index, CCIO_ENABLELOG);
3466     }
3468     rdc_clr_flags(urdc, RDC_SYNCING);
3469     if (rdc_get_vflags(urdc) & RDC_SLAVE) {
3470         rdc_many_enter(krdc);
3471         rdc_clr_mflags(urdc, RDC_SLAVE);
3472         rdc_many_exit(krdc);
3473     }
3474     if (krdc->type_flag & RDC_ASYNCMODE)
3475         rdc_set_flags(urdc, RDC_ASYNC);
3476     if (sync_completed) {
3477         rdc_many_enter(krdc);
3478         rdc_clr_mflags(urdc, RDC_RSYNC_NEEDED);
3479         rdc_many_exit(krdc);
3480     } else {
3481         krdc->remote_index = -1;
3482         rdc_set_flags_log(urdc, RDC_LOGGING, "sync failed to complete");
3483     }
3484     rdc_group_exit(krdc);
3485     rdc_write_state(urdc);
3487     mutex_enter(&net_blk_lock);
3488     if (sync_completed)
3489         krdc->sync_done = RDC_COMPLETED;
3490     else
3491         krdc->sync_done = RDC_FAILED;
3492     cv_broadcast(&krdc->synccv);
3493     mutex_exit(&net_blk_lock);

```

```

3495 }
3498 static int
3499 rdc_sync(rdc_config_t *uparms, spcs_s_info_t kstatus)
3500 {
3501     rdc_set_t *rdc_set = uparms->rdc_set;
3502     int options = uparms->options;
3503     int rc = 0;
3504     int busy = 0;
3505     int index;
3506     rdc_k_info_t *krdc;
3507     rdc_u_info_t *urdc;
3508     rdc_k_info_t *kmulti;
3509     rdc_u_info_t *umulti;
3510     rdc_group_t *group;
3511     rdc_srv_t *svp;
3512     int sm, um, md;
3513     int sync_completed = 0;
3514     int thrcount;
3516     mutex_enter(&rdc_conf_lock);
3517     index = rdc_lookup_byname(rdc_set);
3518     if (index >= 0)
3519         krdc = &rdc_k_info[index];
3520     if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
3521         mutex_exit(&rdc_conf_lock);
3522         spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
3523                 rdc_set->secondary.file);
3524         rc = RDC_EALREADY;
3525         goto notstarted;
3526     }
3528     urdc = &rdc_u_info[index];
3529     group = krdc->group;
3530     set_busy(krdc);
3531     busy = 1;
3532     if ((krdc->type_flag == 0) || (krdc->type_flag & RDC_DISABLEPEND)) {
3533         /* A resume or enable failed or we raced with a teardown */
3534         mutex_exit(&rdc_conf_lock);
3535         spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
3536                 rdc_set->secondary.file);
3537         rc = RDC_EALREADY;
3538         goto notstarted;
3539     }
3540     mutex_exit(&rdc_conf_lock);
3541     rdc_group_enter(krdc);
3543     if (!IS_STATE(urdc, RDC_LOGGING)) {
3544         spcs_s_add(kstatus, RDC_ESETNOTLOGGING, urdc->secondary.intf,
3545                 urdc->secondary.file);
3546         rc = RDC_ENOTLOGGING;
3547         goto notstarted_unlock;
3548     }
3550     if (rdc_check(krdc, rdc_set)) {
3551         spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
3552                 rdc_set->secondary.file);
3553         rc = RDC_EALREADY;
3554         goto notstarted_unlock;
3555     }
3557     if (!(rdc_get_vflags(urdc) & RDC_PRIMARY)) {
3558         spcs_s_add(kstatus, RDC_ENOTPRIMARY, rdc_set->primary.intf,
3559                 rdc_set->primary.file, rdc_set->secondary.intf,

```

```

3560         rdc_set->secondary.file);
3561         rc = RDC_ENOTPRIMARY;
3562         goto notstarted_unlock;
3563     }

3565     if ((options & RDC_OPT_REVERSE) && (IS_STATE(urdc, RDC_QUEUEING))) {
3566         /*
3567          * cannot reverse sync when queuing, need to go logging first
3568          */
3569         spcs_s_add(kstatus, RDC_EQNORSYNC, rdc_set->primary.intf,
3570                 rdc_set->primary.file, rdc_set->secondary.intf,
3571                 rdc_set->secondary.file);
3572         rc = RDC_EQNORSYNC;
3573         goto notstarted_unlock;
3574     }

3576     svp = krdc->lsrv;
3577     krdc->intf = rdc_add_to_if(svp, &(urdc->primary.addr),
3578                             &(urdc->secondary.addr), 1);

3580     if (!krdc->intf) {
3581         spcs_s_add(kstatus, RDC_EADDTOIF, urdc->primary.intf,
3582                 urdc->secondary.intf);
3583         rc = RDC_EADDTOIF;
3584         goto notstarted_unlock;
3585     }

3587     if (urdc->volume_size == 0) {
3588         /* Implies reserve failed when previous resume was done */
3589         rdc_get_details(krdc);
3590     }
3591     if (urdc->volume_size == 0) {
3592         spcs_s_add(kstatus, RDC_ENOEMAP);
3593         rc = RDC_ENOEMAP;
3594         goto notstarted_unlock;
3595     }

3597     if (krdc->dcio_bitmap == NULL) {
3598         if (rdc_resume_bitmap(krdc) < 0) {
3599             spcs_s_add(kstatus, RDC_ENOEMAP);
3600             rc = RDC_ENOEMAP;
3601             goto notstarted_unlock;
3602         }
3603     }

3605     if ((rdc_get_vflags(urdc) & RDC_BMP_FAILED) && (krdc->bitmapfd)) {
3606         if (rdc_reset_bitmap(krdc) {
3607             spcs_s_add(kstatus, RDC_EBITMAP);
3608             rc = RDC_EBITMAP;
3609             goto notstarted_unlock;
3610         }
3611     }

3613     if (IS_MANY(krdc) || IS_MULTI(krdc)) {
3614         rdc_u_info_t *ubad;

3616         if ((ubad = rdc_allow_pri_sync(urdc, options)) != NULL) {
3617             spcs_s_add(kstatus, RDC_ESTATE,
3618                     ubad->primary.intf, ubad->primary.file,
3619                     ubad->secondary.intf, ubad->secondary.file);
3620             rc = RDC_ESTATE;
3621             goto notstarted_unlock;
3622         }
3623     }

3625     /*

```

```

3626         * there is a small window where _rdc_sync is still
3627         * running, but has cleared the RDC_SYNCING flag.
3628         * Use aux_state which is only cleared
3629         * after _rdc_sync had done its 'death' broadcast.
3630         */
3631         if (krdc->aux_state & RDC_AUXSYNCIP) {
3632             #ifdef DEBUG
3633                 if (!rdc_get_vflags(urdc) & RDC_SYNCING) {
3634                     cmn_err(CE_WARN, "rdc_sync: "
3635                             "RDC_AUXSYNCIP set, SYNCING off");
3636                 }
3637             #endif
3638             spcs_s_add(kstatus, RDC_ESYNCING, rdc_set->primary.file);
3639             rc = RDC_ESYNCING;
3640             goto notstarted_unlock;
3641         }
3642         if (krdc->disk_status == 1) {
3643             spcs_s_add(kstatus, RDC_ESYNCING, rdc_set->primary.file);
3644             rc = RDC_ESYNCING;
3645             goto notstarted_unlock;
3646         }

3648         if ((options & RDC_OPT_FORWARD) &&
3649             (rdc_get_mflags(urdc) & RDC_RSYNC_NEEDED)) {
3650             /* cannot forward sync if a reverse sync is needed */
3651             spcs_s_add(kstatus, RDC_ESYNCNEEDED, rdc_set->primary.intf,
3652                     rdc_set->primary.file, rdc_set->secondary.intf,
3653                     rdc_set->secondary.file);
3654             rc = RDC_ESYNCNEEDED;
3655             goto notstarted_unlock;
3656         }

3658         urdc->sync_pos = 0;

3660         /* Check if the rdc set is accessible on the remote node */
3661         if (rdc_net_getstate(krdc, &sm, &um, &md, FALSE) < 0) {
3662             /*
3663              * Remote end may be inaccessible, or the rdc set is not
3664              * enabled at the remote end.
3665              */
3666             spcs_s_add(kstatus, RDC_ECONNOPEN, urdc->secondary.intf,
3667                     urdc->secondary.file);
3668             rc = RDC_ECONNOPEN;
3669             goto notstarted_unlock;
3670         }
3671         if (options & RDC_OPT_REVERSE)
3672             krdc->remote_index = rdc_net_state(index, CCIO_RSYNC);
3673         else
3674             krdc->remote_index = rdc_net_state(index, CCIO_SLAVE);
3675         if (krdc->remote_index < 0) {
3676             /*
3677              * Remote note probably not in a valid state to be synced,
3678              * as the state was fetched OK above.
3679              */
3680             spcs_s_add(kstatus, RDC_ERSTATE, urdc->secondary.intf,
3681                     urdc->secondary.file, urdc->primary.intf,
3682                     urdc->primary.file);
3683             rc = RDC_ERSTATE;
3684             goto notstarted_unlock;
3685         }

3687         rc = check_filesize(index, kstatus);
3688         if (rc != 0) {
3689             (void) rdc_net_state(krdc->index, CCIO_ENABLELOG);
3690             goto notstarted_unlock;
3691         }

```

```

3693     krdc->sync_done = 0;

3695     mutex_enter(&krdc->bmapmutex);
3696     krdc->aux_state |= RDC_AUXSYNCIP;
3697     mutex_exit(&krdc->bmapmutex);

3699     if (options & RDC_OPT_REVERSE) {
3700         rdc_many_enter(krdc);
3701         rdc_set_mflags(urdc, RDC_SLAVE | RDC_RSYNC_NEEDED);
3702         mutex_enter(&krdc->bmapmutex);
3703         rdc_clr_flags(urdc, RDC_VOL_FAILED);
3704         mutex_exit(&krdc->bmapmutex);
3705         rdc_write_state(urdc);
3706         /* LINTED */
3707         if (kmulti = krdc->multi_next) {
3708             umulti = &rdc_u_info[kmulti->index];
3709             if (IS_ENABLED(umulti) && (rdc_get_vflags(umulti) &
3710                 (RDC_VOL_FAILED | RDC_SYNC_NEEDED))) {
3711                 rdc_clr_flags(umulti, RDC_SYNC_NEEDED);
3712                 rdc_clr_flags(umulti, RDC_VOL_FAILED);
3713                 rdc_write_state(umulti);
3714             }
3715         }
3716         rdc_many_exit(krdc);
3717     } else {
3718         rdc_clr_flags(urdc, RDC_FCAL_FAILED);
3719         rdc_write_state(urdc);
3720     }

3722     if (options & RDC_OPT_UPDATE) {
3723         ASSERT(urdc->volume_size != 0);
3724         if (rdc_net_getbmap(index,
3725             BMAP_LOG_BYTES(urdc->volume_size)) > 0) {
3726             spcs_s_add(kstatus, RDC_ENOBMAP);
3727             rc = RDC_ENOBMAP;
3729             (void) rdc_net_state(index, CCIO_ENABLELOG);

3731             rdc_clr_flags(urdc, RDC_SYNCING);
3732             if (options & RDC_OPT_REVERSE) {
3733                 rdc_many_enter(krdc);
3734                 rdc_clr_mflags(urdc, RDC_SLAVE);
3735                 rdc_many_exit(krdc);
3736             }
3737             if (krdc->type_flag & RDC_ASYNCMODE)
3738                 rdc_set_flags(urdc, RDC_ASYNC);
3739             krdc->remote_index = -1;
3740             rdc_set_flags_log(urdc, RDC_LOGGING,
3741                 "failed to read remote bitmap");
3742             rdc_write_state(urdc);
3743             goto failed;
3744         }
3745         rdc_clr_flags(urdc, RDC_FULL);
3746     } else {
3747         /*
3748          * This is a full sync (not an update sync), mark the
3749          * entire bitmap dirty
3750          */
3751         (void) RDC_FILL_BITMAP(krdc, FALSE);

3753         rdc_set_flags(urdc, RDC_FULL);
3754     }

3756     rdc_group_exit(krdc);

```

```

3758     /*
3759     * allow diskq->memq flusher to wake up
3760     */
3761     mutex_enter(&krdc->group->ra_queue.net_qlock);
3762     krdc->group->ra_queue.qfflags &= ~RDC_QFILLSLEEP;
3763     mutex_exit(&krdc->group->ra_queue.net_qlock);

3765     /*
3766     * if this is a full sync on a non-diskq set or
3767     * a diskq set that has failed, clear the async flag
3768     */
3769     if (krdc->type_flag & RDC_ASYNCMODE) {
3770         if (((options & RDC_OPT_UPDATE) ||
3771             (!RDC_IS_DISKQ(krdc->group)) ||
3772             (!IS_STATE(urdc, RDC_QUEUING)))) {
3773             /* full syncs, or core queue are synchronous */
3774             rdc_group_enter(krdc);
3775             rdc_clr_flags(urdc, RDC_ASYNC);
3776             rdc_group_exit(krdc);
3777         }

3779         /*
3780         * if the queue failed because it was full, lets see
3781         * if we can restart it. After _rdc_sync() is done
3782         * the modes will switch and we will begin disk
3783         * queuing again. NOTE: this should only be called
3784         * once per group, as it clears state for all group
3785         * members, also clears the async flag for all members
3786         */
3787         if (IS_STATE(urdc, RDC_DISK_FAILED)) {
3788             rdc_unfail_diskq(krdc);
3789         } else {
3790             /* don't add insult to injury by flushing a dead queue */

3792             /*
3793             * if we are updating, and a diskq and
3794             * the async thread isn't active, start
3795             * it up.
3796             */
3797             if ((options & RDC_OPT_UPDATE) &&
3798                 (IS_STATE(urdc, RDC_QUEUING))) {
3799                 rdc_group_enter(krdc);
3800                 rdc_clr_flags(urdc, RDC_SYNCING);
3801                 rdc_group_exit(krdc);
3802                 mutex_enter(&krdc->group->ra_queue.net_qlock);
3803                 if (krdc->group->ra_queue.qfill_sleeping ==
3804                     RDC_QFILL_ASLEEP)
3805                     cv_broadcast(&group->ra_queue.qfcv);
3806                 mutex_exit(&krdc->group->ra_queue.net_qlock);
3807                 thrcount = urdc->asyncthr;
3808                 while ((thrcount-- > 0) &&
3809                     !krdc->group->rdc_writer) {
3810                     (void) rdc_writer(krdc->index);
3811                 }
3812             }
3813         }
3814     }

3816     /*
3817     * For a reverse sync, merge the current bitmap with all other sets
3818     * that share this volume.
3819     */
3820     if (options & RDC_OPT_REVERSE) {
3821     retry_many:
3822         rdc_many_enter(krdc);
3823         if (IS_MANY(krdc)) {

```



```

3824         rdc_k_info_t *kmany;
3825         rdc_u_info_t *umany;

3827         for (kmany = krdc->many_next; kmany != krdc;
3828             kmany = kmany->many_next) {
3829             umany = &rdc_u_info[kmany->index];
3830             if (!IS_ENABLED(umany))
3831                 continue;
3832             ASSERT(umany->flags & RDC_PRIMARY);

3834             if (!mutex_tryenter(&kmany->group->lock)) {
3835                 rdc_many_exit(krdc);
3836                 /* May merge more than once */
3837                 goto retry_many;
3838             }
3839             rdc_merge_bitmaps(krdc, kmany);
3840             mutex_exit(&kmany->group->lock);
3841         }
3842     }
3843     rdc_many_exit(krdc);

3845 retry_multi:
3846     rdc_many_enter(krdc);
3847     if (IS_MULTII(krdc)) {
3848         rdc_k_info_t *kmulti = krdc->multi_next;
3849         rdc_u_info_t *umulti = &rdc_u_info[kmulti->index];

3851         if (IS_ENABLED(umulti)) {
3852             ASSERT(!(umulti->flags & RDC_PRIMARY));

3854             if (!mutex_tryenter(&kmulti->group->lock)) {
3855                 rdc_many_exit(krdc);
3856                 goto retry_multi;
3857             }
3858             rdc_merge_bitmaps(krdc, kmulti);
3859             mutex_exit(&kmulti->group->lock);
3860         }
3861     }
3862     rdc_many_exit(krdc);
3863 }

3865 rdc_group_enter(krdc);

3867 if (krdc->bitmap_write == 0) {
3868     if (rdc_write_bitmap_fill(krdc) >= 0)
3869         krdc->bitmap_write = -1;
3870 }

3872 if (krdc->bitmap_write > 0)
3873     (void) rdc_write_bitmap(krdc);

3875 urdc->bits_set = RDC_COUNT_BITMAP(krdc);
3877 rdc_group_exit(krdc);

3879 if (options & RDC_OPT_REVERSE) {
3880     (void) _rdc_sync_event_notify(RDC_SYNC_START,
3881     urdc->primary.file, urdc->group_name);
3882 }

3884 /* Now set off the sync itself */

3886 mutex_enter(&net_blk_lock);
3887 if (nsc_create_process(
3888     (void (*)(void *))_rdc_sync, (void *)krdc, FALSE)) {
3889     mutex_exit(&net_blk_lock);

```

```

3890         spcs_s_add(kstatus, RDC_ENOPROC);
3891         /*
3892          * We used to just return here,
3893          * but we need to clear the AUXSYNCIP bit
3894          * and there is a very small chance that
3895          * someone may be waiting on the disk_status flag.
3896          */
3897         rc = RDC_ENOPROC;
3898         /*
3899          * need the group lock held at failed.
3900          */
3901         rdc_group_enter(krdc);
3902         goto failed;
3903     }

3905     mutex_enter(&rdc_conf_lock);
3906     wakeup_busy(krdc);
3907     busy = 0;
3908     mutex_exit(&rdc_conf_lock);

3910     while (krdc->sync_done == 0)
3911         cv_wait(&krdc->sync_cv, &net_blk_lock);
3912     mutex_exit(&net_blk_lock);

3914     rdc_group_enter(krdc);

3916     if (krdc->sync_done == RDC_FAILED) {
3917         char siztmp1[16];
3918         (void) spcs_s_inttostring(
3919             urdc->sync_pos, siztmp1, sizeof (siztmp1),
3920             0);
3921         spcs_s_add(kstatus, RDC_EFAIL, siztmp1);
3922         rc = RDC_EFAIL;
3923     } else
3924         sync_completed = 1;

3926 failed:
3927     /*
3928      * We use this flag now to make halt_sync() wait for
3929      * us to terminate and let us take the group lock.
3930      */
3931     krdc->aux_state && ~RDC_AUXSYNCIP;
3932     if (krdc->disk_status == 1) {
3933         krdc->disk_status = 0;
3934         cv_broadcast(&krdc->haltcv);
3935     }

3937 notstarted_unlock:
3938     rdc_group_exit(krdc);

3940     if (sync_completed && (options & RDC_OPT_REVERSE)) {
3941         (void) _rdc_sync_event_notify(RDC_SYNC_DONE,
3942         urdc->primary.file, urdc->group_name);
3943     }

3945 notstarted:
3946     if (busy) {
3947         mutex_enter(&rdc_conf_lock);
3948         wakeup_busy(krdc);
3949         mutex_exit(&rdc_conf_lock);
3950     }

3952     return (rc);
3953 }

3955 /* ARGSUSED */

```

```

3956 static int
3957 rdc_suspend(rdc_k_info_t *krdc, rdc_set_t *rdc_set, spcs_s_info_t kstatus)
3958 {
3959     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
3960     rdc_if_t *ip;
3961     int index = krdc->index;

3963     ASSERT(krdc->group != NULL);
3964     rdc_group_enter(krdc);
3965 #ifdef DEBUG
3966     ASSERT(rdc_check(krdc, rdc_set) == 0);
3967 #else
3968     if (rdc_check(krdc, rdc_set)) {
3969         rdc_group_exit(krdc);
3970         spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
3971                 rdc_set->secondary.file);
3972         return (RDC_EALREADY);
3973     }
3974 #endif

3976     if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
3977         halt_sync(krdc);
3978         ASSERT(IS_ENABLED(urdc));
3979     }

3981     rdc_group_exit(krdc);
3982     (void) rdc_unintercept(krdc);

3984 #ifdef DEBUG
3985     cmn_err(CE_NOTE, "!SNDR: suspended %s %s", urdc->primary.file,
3986            urdc->secondary.file);
3987 #endif

3989     /* Configured but not enabled */
3990     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

3993     if (IS_ASYNC(urdc) && !RDC_IS_DISKQ(krdc->group)) {
3994         int tries = 2; /* in case of possibly stuck flusher threads */
3995 #ifdef DEBUG
3996         net_queue *qp = &krdc->group->ra_queue;
3997 #endif
3998         do {
3999             if (!krdc->group->rdc_writer)
4000                 (void) rdc_writer(krdc->index);

4002                 (void) rdc_drain_queue(krdc->index);

4004         } while (krdc->group->rdc_writer && tries--);

4006     /* ok, force it to happen... */
4007     if (rdc_drain_queue(krdc->index) != 0) {
4008         do {
4009             mutex_enter(&krdc->group->ra_queue.net_qlock);
4010             krdc->group->asynccdis = 1;
4011             cv_broadcast(&krdc->group->asynccqv);
4012             mutex_exit(&krdc->group->ra_queue.net_qlock);
4013             cmn_err(CE_WARN,
4014                    "!SNDR: async I/O pending and not flushed "
4015                    "for %s during suspend",
4016                    urdc->primary.file);
4017 #ifdef DEBUG
4018             cmn_err(CE_WARN,
4019                    "!nitems: %" NSC_SZFMT " nblocks: %"
4020                    NSC_SZFMT " head: 0x%p tail: 0x%p",
4021                    qp->nitems, qp->nblocks,

```

```

4022             (void *)qp->net_qhead,
4023             (void *)qp->net_qtail);
4024 #endif
4025         } while (krdc->group->rdc_thrnum > 0);
4026     }
4027 }

4029     mutex_enter(&rdc_conf_lock);
4030     ip = krdc->intf;
4031     krdc->intf = 0;

4033     if (ip) {
4034         rdc_remove_from_if(ip);
4035     }

4037     mutex_exit(&rdc_conf_lock);

4039     rdc_group_enter(krdc);

4041     /* Configured but not enabled */
4042     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

4044     rdc_group_exit(krdc);
4045     /* Must not hold group lock during this function */
4046     while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
4047         delay(2);
4048     rdc_group_enter(krdc);

4050     /* Don't rdc_clear_state, unlike _rdc_disable */

4052     rdc_free_bitmap(krdc, RDC_CMD_SUSPEND);
4053     rdc_close_bitmap(krdc);

4055     rdc_dev_close(krdc);
4056     rdc_close_direct(krdc);

4058     /* Configured but not enabled */
4059     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

4061     rdc_group_exit(krdc);

4063     /*
4064     * we should now unregister the queue, with no conflicting
4065     * locks held. This is the last(only) member of the group
4066     */
4067     if (krdc->group && RDC_IS_DISKQ(krdc->group) &&
4068         krdc->group->count == 1) { /* stop protecting queue */
4069         rdc_unintercept_diskq(krdc->group);
4070     }

4072     mutex_enter(&rdc_conf_lock);

4074     /* Configured but not enabled */
4075     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));

4077     wait_busy(krdc);

4079     if (IS_MANY(krdc) || IS_MULTI(krdc))
4080         remove_from_many(krdc);

4082     remove_from_group(krdc);

4084     krdc->remote_index = -1;
4085     ASSERT(krdc->type_flag & RDC_CONFIGURED);
4086     ASSERT(krdc->type_flag & RDC_DISABLEPEND);
4087     krdc->type_flag = 0;

```

```

4088 #ifdef DEBUG
4089     if (krdc->dcio_bitmap)
4090         cmn_err(CE_WARN, "!_rdc_suspend: possible mem leak, "
4091             "dcio_bitmap");
4092 #endif
4093     krdc->dcio_bitmap = NULL;
4094     krdc->bitmap_ref = NULL;
4095     krdc->bitmap_size = 0;
4096     krdc->maxfbas = 0;
4097     krdc->bitmap_write = 0;
4098     krdc->disk_status = 0;
4099     rdc_destroy_svinfo(krdc->lsrv);
4100     krdc->lsrv = NULL;
4101     krdc->multi_next = NULL;
4102
4103     rdc_u_init(urdc);
4104
4105     mutex_exit(&rdc_conf_lock);
4106     rdc_kstat_delete(index);
4107     return (0);
4108 }
4109
4110 static int
4111 rdc_suspend(rdc_config_t *uparms, spcs_s_info_t kstatus)
4112 {
4113     rdc_k_info_t *krdc;
4114     int index;
4115     int rc;
4116
4117     mutex_enter(&rdc_conf_lock);
4118
4119     index = rdc_lookup_byname(uparms->rdc_set);
4120     if (index >= 0)
4121         krdc = &rdc_k_info[index];
4122     if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
4123         mutex_exit(&rdc_conf_lock);
4124         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
4125             uparms->rdc_set->secondary.file);
4126         return (RDC_EALREADY);
4127     }
4128
4129     krdc->type_flag |= RDC_DISABLEPEND;
4130     wait_busy(krdc);
4131     if (krdc->type_flag == 0) {
4132         /* A resume or enable failed */
4133         mutex_exit(&rdc_conf_lock);
4134         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
4135             uparms->rdc_set->secondary.file);
4136         return (RDC_EALREADY);
4137     }
4138     mutex_exit(&rdc_conf_lock);
4139
4140     rc = _rdc_suspend(krdc, uparms->rdc_set, kstatus);
4141     return (rc);
4142 }
4143
4144 static int
4145 _rdc_resume(rdc_set_t *rdc_set, int options, spcs_s_info_t kstatus)
4146 {
4147     int index;
4148     char *rhost;
4149     struct netbuf *addrp;
4150     rdc_k_info_t *krdc;
4151     rdc_u_info_t *urdc;
4152     rdc_srv_t *svp = NULL;
4153     char *local_file;

```

```

4154     char *local_bitmap;
4155     int rc, rcl;
4156     nsc_size_t maxfbas;
4157     rdc_group_t *grp;
4158
4159     if ((rdc_set->primary.intf[0] == 0) ||
4160         (rdc_set->primary.addr.len == 0) ||
4161         (rdc_set->primary.file[0] == 0) ||
4162         (rdc_set->primary.bitmap[0] == 0) ||
4163         (rdc_set->secondary.intf[0] == 0) ||
4164         (rdc_set->secondary.addr.len == 0) ||
4165         (rdc_set->secondary.file[0] == 0) ||
4166         (rdc_set->secondary.bitmap[0] == 0)) {
4167         spcs_s_add(kstatus, RDC_EEMPTY);
4168         return (RDC_EEMPTY);
4169     }
4170
4171     /* Next check there aren't any enabled rdc sets which match. */
4172
4173     mutex_enter(&rdc_conf_lock);
4174
4175     if (rdc_lookup_byname(rdc_set) >= 0) {
4176         mutex_exit(&rdc_conf_lock);
4177         spcs_s_add(kstatus, RDC_EENABLED, rdc_set->primary.intf,
4178             rdc_set->primary.file, rdc_set->secondary.intf,
4179             rdc_set->secondary.file);
4180         return (RDC_EENABLED);
4181     }
4182
4183     if (rdc_lookup_many2one(rdc_set) >= 0) {
4184         mutex_exit(&rdc_conf_lock);
4185         spcs_s_add(kstatus, RDC_EMANY2ONE, rdc_set->primary.intf,
4186             rdc_set->primary.file, rdc_set->secondary.intf,
4187             rdc_set->secondary.file);
4188         return (RDC_EMANY2ONE);
4189     }
4190
4191     if (rdc_set->netconfig->knc_proto == NULL) {
4192         mutex_exit(&rdc_conf_lock);
4193         spcs_s_add(kstatus, RDC_ENETCONFIG);
4194         return (RDC_ENETCONFIG);
4195     }
4196
4197     if (rdc_set->primary.addr.len == 0) {
4198         mutex_exit(&rdc_conf_lock);
4199         spcs_s_add(kstatus, RDC_ENETBUF, rdc_set->primary.file);
4200         return (RDC_ENETBUF);
4201     }
4202
4203     if (rdc_set->secondary.addr.len == 0) {
4204         mutex_exit(&rdc_conf_lock);
4205         spcs_s_add(kstatus, RDC_ENETBUF, rdc_set->secondary.file);
4206         return (RDC_ENETBUF);
4207     }
4208
4209     /* Check that the local data volume isn't in use as a bitmap */
4210     if (options & RDC_OPT_PRIMARY)
4211         local_file = rdc_set->primary.file;
4212     else
4213         local_file = rdc_set->secondary.file;
4214     if (rdc_lookup_bitmap(local_file) >= 0) {
4215         mutex_exit(&rdc_conf_lock);
4216         spcs_s_add(kstatus, RDC_EVOLINUSE, local_file);
4217         return (RDC_EVOLINUSE);
4218     }

```

```

4220 /* check that the secondary data volume isn't in use */
4221 if (!(options & RDC_OPT_PRIMARY)) {
4222     local_file = rdc_set->secondary.file;
4223     if (rdc_lookup_secondary(local_file) >= 0) {
4224         mutex_exit(&rdc_conf_lock);
4225         spcs_s_add(kstatus, RDC_EVOLINUSE, local_file);
4226         return (RDC_EVOLINUSE);
4227     }
4228 }
4230 /* Check that the bitmap isn't in use as a data volume */
4231 if (options & RDC_OPT_PRIMARY)
4232     local_bitmap = rdc_set->primary.bitmap;
4233 else
4234     local_bitmap = rdc_set->secondary.bitmap;
4235 if (rdc_lookup_configured(local_bitmap) >= 0) {
4236     mutex_exit(&rdc_conf_lock);
4237     spcs_s_add(kstatus, RDC_EBMPINUSE, local_bitmap);
4238     return (RDC_EBMPINUSE);
4239 }
4241 /* Check that the bitmap isn't already in use as a bitmap */
4242 if (rdc_lookup_bitmap(local_bitmap) >= 0) {
4243     mutex_exit(&rdc_conf_lock);
4244     spcs_s_add(kstatus, RDC_EBMPINUSE, local_bitmap);
4245     return (RDC_EBMPINUSE);
4246 }
4248 /* Set urdc->volume_size */
4249 index = rdc_dev_open(rdc_set, options);
4250 if (index < 0) {
4251     mutex_exit(&rdc_conf_lock);
4252     if (options & RDC_OPT_PRIMARY)
4253         spcs_s_add(kstatus, RDC_EOPEN, rdc_set->primary.intf,
4254                 rdc_set->primary.file);
4255     else
4256         spcs_s_add(kstatus, RDC_EOPEN, rdc_set->secondary.intf,
4257                 rdc_set->secondary.file);
4258     return (RDC_EOPEN);
4259 }
4261 urdc = &rdc_u_info[index];
4262 krdc = &rdc_k_info[index];
4264 /* copy relevant parts of rdc_set to urdc field by field */
4266 (void) strncpy(urdc->primary.intf, rdc_set->primary.intf,
4267             MAX_RDC_HOST_SIZE);
4268 (void) strncpy(urdc->secondary.intf, rdc_set->secondary.intf,
4269             MAX_RDC_HOST_SIZE);
4271 (void) strncpy(urdc->group_name, rdc_set->group_name, NSC_MAXPATH);
4273 dup_rdc_netbuf(&rdc_set->primary.addr, &urdc->primary.addr);
4274 (void) strncpy(urdc->primary.file, rdc_set->primary.file, NSC_MAXPATH);
4275 (void) strncpy(urdc->primary.bitmap, rdc_set->primary.bitmap,
4276             NSC_MAXPATH);
4278 dup_rdc_netbuf(&rdc_set->secondary.addr, &urdc->secondary.addr);
4279 (void) strncpy(urdc->secondary.file, rdc_set->secondary.file,
4280             NSC_MAXPATH);
4281 (void) strncpy(urdc->secondary.bitmap, rdc_set->secondary.bitmap,
4282             NSC_MAXPATH);
4283 (void) strncpy(urdc->disk_queue, rdc_set->disk_queue, NSC_MAXPATH);
4284 urdc->setid = rdc_set->setid;

```

```

4286 if ((options & RDC_OPT_SYNC) && urdc->disk_queue[0]) {
4287     mutex_exit(&rdc_conf_lock);
4288     rdc_dev_close(krdc);
4289     spcs_s_add(kstatus, RDC_EQWRONGMODE);
4290     return (RDC_EQWRONGMODE);
4291 }
4293 /*
4294  * init flags now so that state left by failures in add_to_group()
4295  * are preserved.
4296  */
4297 rdc_init_flags(urdc);
4299 if ((rc1 = add_to_group(krdc, options, RDC_CMD_RESUME)) != 0) {
4300     if (rc1 == RDC_EQNOADD) { /* something went wrong with queue */
4301         rdc_fail_diskq(krdc, RDC_WAIT, RDC_NOLOG);
4302         /* don't return a failure here, continue with resume */
4304     } else { /* some other group add failure */
4305         mutex_exit(&rdc_conf_lock);
4306         rdc_dev_close(krdc);
4307         spcs_s_add(kstatus, RDC_EGROUP,
4308                 rdc_set->primary.intf, rdc_set->primary.file,
4309                 rdc_set->secondary.intf, rdc_set->secondary.file,
4310                 rdc_set->group_name);
4311         return (RDC_EGROUP);
4312     }
4313 }
4315 /*
4316  * maxfbas was set in rdc_dev_open as primary's maxfbas.
4317  * If diskq's maxfbas is smaller, then use diskq's.
4318  */
4319 grp = krdc->group;
4320 if (grp && RDC_IS_DISKQ(grp) && (grp->diskqfd != 0)) {
4321     rc = _rdc_rsrv_diskq(grp);
4322     if (RDC_SUCCESS(rc)) {
4323         rc = nsc_maxfbas(grp->diskqfd, 0, &maxfbas);
4324         if (rc == 0) {
4325 #ifdef DEBUG
4326             if (krdc->maxfbas != maxfbas)
4327                 cmn_err(CE_NOTE,
4328                     "!_rdc_resume: diskq maxfbas = %"
4329                     NSC_SZFMT ", primary maxfbas = %"
4330                     NSC_SZFMT, maxfbas, krdc->maxfbas);
4331 #endif
4332             krdc->maxfbas = min(krdc->maxfbas,
4333                             maxfbas);
4334         } else {
4335             cmn_err(CE_WARN,
4336                 "!_rdc_resume: diskq maxfbas failed (%d)",
4337                 rc);
4338         }
4339     }
4340 } else {
4341     cmn_err(CE_WARN,
4342         "!_rdc_resume: diskq reserve failed (%d)", rc);
4343 }
4344 }
4346 (void) strncpy(urdc->direct_file, rdc_set->direct_file, NSC_MAXPATH);
4347 if ((options & RDC_OPT_PRIMARY) && rdc_set->direct_file[0]) {
4348     if (rdc_open_direct(krdc) == NULL)
4349         rdc_set_flags(urdc, RDC_FCAL_FAILED);
4350 }

```

```

4352     krdc->many_next = krdc;
4354     ASSERT(krdc->type_flag == 0);
4355     krdc->type_flag = RDC_CONFIGURED;
4357     if (options & RDC_OPT_PRIMARY)
4358         rdc_set_flags(urdc, RDC_PRIMARY);
4360     if (options & RDC_OPT_ASYNC)
4361         krdc->type_flag |= RDC_ASYNCMODE;
4363     set_busy(krdc);
4365     urdc->syshostid = rdc_set->syshostid;
4367     if (add_to_many(krdc) < 0) {
4368         mutex_exit(&rdc_conf_lock);
4370         rdc_group_enter(krdc);
4372         spcs_s_add(kstatus, RDC_EMULTI);
4373         rc = RDC_EMULTI;
4374         goto fail;
4375     }
4377     /* Configured but not enabled */
4378     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));
4380     mutex_exit(&rdc_conf_lock);
4382     if (urdc->volume_size == 0) {
4383         rdc_many_enter(krdc);
4384         if (options & RDC_OPT_PRIMARY)
4385             rdc_set_mflags(urdc, RDC_RSYNC_NEEDED);
4386         else
4387             rdc_set_flags(urdc, RDC_SYNC_NEEDED);
4388         rdc_set_flags(urdc, RDC_VOL_FAILED);
4389         rdc_many_exit(krdc);
4390     }
4392     rdc_group_enter(krdc);
4394     /* Configured but not enabled */
4395     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));
4397     /*
4398     * The rdc set is configured but not yet enabled. Other operations must
4399     * ignore this set until it is enabled.
4400     */
4402     urdc->sync_pos = 0;
4404     /* Set tunable defaults, we'll pick up tunables from the header later */
4406     urdc->maxqfbas = rdc_maxthres_queue;
4407     urdc->maxqitems = rdc_max_qitems;
4408     urdc->autosync = 0;
4409     urdc->asyncthr = rdc_asyncthr;
4411     urdc->netconfig = rdc_set->netconfig;
4413     if (options & RDC_OPT_PRIMARY) {
4414         rhost = rdc_set->secondary.intf;
4415         addrp = &rdc_set->secondary.addr;
4416     } else {
4417         rhost = rdc_set->primary.intf;

```

```

4418         addrp = &rdc_set->primary.addr;
4419     }
4421     if (options & RDC_OPT_ASYNC)
4422         rdc_set_flags(urdc, RDC_ASYNC);
4424     svp = rdc_create_svinfo(rhost, addrp, urdc->netconfig);
4425     if (svp == NULL) {
4426         spcs_s_add(kstatus, ENOMEM);
4427         rc = ENOMEM;
4428         goto fail;
4429     }
4431     urdc->netconfig = NULL; /* This will be no good soon */
4433     /* Don't set krdc->intf here */
4434     rdc_kstat_create(index);
4436     /* if the bitmap resume isn't clean, it will clear queuing flag */
4438     (void) rdc_resume_bitmap(krdc);
4440     if (RDC_IS_DISKQ(krdc->group)) {
4441         disk_queue *q = &krdc->group->diskq;
4442         if ((rc1 == RDC_EQNOADD) ||
4443             IS_QSTATE(q, RDC_QBADRESUME)) {
4444             rdc_clr_flags(urdc, RDC_QUEUING);
4445             RDC_ZERO_BITREF(krdc);
4446         }
4447     }
4449     if (krdc->lsrv == NULL)
4450         krdc->lsrv = svp;
4451     else {
4452 #ifdef DEBUG
4453         cmn_err(CE_WARN, "!rdc_resume: krdc->lsrv already set: %p",
4454             (void *) krdc->lsrv);
4455 #endif
4456         rdc_destroy_svinfo(svp);
4457     }
4458     svp = NULL;
4460     /* Configured but not enabled */
4461     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));
4463     /* And finally */
4465     krdc->remote_index = -1;
4467     /* Should we set the whole group logging? */
4468     rdc_set_flags(urdc, RDC_ENABLED | RDC_LOGGING);
4470     rdc_group_exit(krdc);
4472     if (rdc_intercept(krdc) != 0) {
4473         rdc_group_enter(krdc);
4474         rdc_clr_flags(urdc, RDC_ENABLED);
4475         if (options & RDC_OPT_PRIMARY)
4476             spcs_s_add(kstatus, RDC_EREGISTER, urdc->primary.file);
4477         else
4478             spcs_s_add(kstatus, RDC_EREGISTER,
4479                 urdc->secondary.file);
4480 #ifdef DEBUG
4481         cmn_err(CE_NOTE, "!nsc_register_path failed %s",
4482             urdc->primary.file);
4483 #endif

```

```

4484         rc = RDC_EREGISTER;
4485         goto bmpfail;
4486     }
4487 #ifdef DEBUG
4488     cmn_err(CE_NOTE, "!SNDR: resumed %s %s", urdc->primary.file,
4489           urdc->secondary.file);
4490 #endif
4492     rdc_write_state(urdc);
4494     mutex_enter(&rdc_conf_lock);
4495     wakeup_busy(krdc);
4496     mutex_exit(&rdc_conf_lock);
4498     return (0);
4500 bmpfail:
4501     if (options & RDC_OPT_PRIMARY)
4502         spcs_s_add(kstatus, RDC_EBITMAP, urdc->primary.bitmap);
4503     else
4504         spcs_s_add(kstatus, RDC_EBITMAP, urdc->secondary.bitmap);
4505     rc = RDC_EBITMAP;
4506     if (rdc_get_vflags(urdc) & RDC_ENABLED) {
4507         rdc_group_exit(krdc);
4508         (void) rdc_unintercept(krdc);
4509         rdc_group_enter(krdc);
4510     }
4512 fail:
4513     rdc_kstat_delete(index);
4514     /* Don't unset krdc->intf here, unlike _rdc_enable */
4516     /* Configured but not enabled */
4517     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));
4519     rdc_dev_close(krdc);
4520     rdc_close_direct(krdc);
4521     rdc_destroy_svinfo(svp);
4523     /* Configured but not enabled */
4524     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));
4526     rdc_group_exit(krdc);
4528     mutex_enter(&rdc_conf_lock);
4530     /* Configured but not enabled */
4531     ASSERT(IS_CONFIGURED(krdc) && !IS_ENABLED(urdc));
4533     remove_from_group(krdc);
4535     if (IS_MANY(krdc) || IS_MULTI(krdc))
4536         remove_from_many(krdc);
4538     rdc_u_init(urdc);
4540     ASSERT(krdc->type_flag & RDC_CONFIGURED);
4541     krdc->type_flag = 0;
4542     wakeup_busy(krdc);
4544     mutex_exit(&rdc_conf_lock);
4546     return (rc);
4547 }
4549 static int

```

```

4550 rdc_resume(rdc_config_t *uparms, spcs_s_info_t kstatus)
4551 {
4552     char itmp[10];
4553     int rc;
4555     if (!(uparms->options & RDC_OPT_SYNC) &&
4556         !(uparms->options & RDC_OPT_ASYNC)) {
4557         (void) spcs_s_inttostring(
4558             uparms->options, itmp, sizeof(itmp), 1);
4559         spcs_s_add(kstatus, RDC_EEINVAL, itmp);
4560         rc = RDC_EEINVAL;
4561         goto done;
4562     }
4564     if (!(uparms->options & RDC_OPT_PRIMARY) &&
4565         !(uparms->options & RDC_OPT_SECONDARY)) {
4566         (void) spcs_s_inttostring(
4567             uparms->options, itmp, sizeof(itmp), 1);
4568         spcs_s_add(kstatus, RDC_EEINVAL, itmp);
4569         rc = RDC_EEINVAL;
4570         goto done;
4571     }
4573     rc = _rdc_resume(uparms->rdc_set, uparms->options, kstatus);
4574 done:
4575     return (rc);
4576 }
4578 /*
4579  * if rdc_group_log is called because a volume has failed,
4580  * we must discard the queue to preserve write ordering.
4581  * later perhaps, we can keep queuing, but we would have to
4582  * rewrite the i/o path to accommodate that. currently, if there
4583  * is a volume failure, the buffers are satisfied remotely and
4584  * there is no way to satisfy them from the current diskq config
4585  * phew, if we do that.. it will be difficult
4586  */
4587 int
4588 rdc_can_queue(rdc_k_info_t *krdc)
4589 {
4590     rdc_k_info_t *p;
4591     rdc_u_info_t *q;
4593     for (p = krdc->group_next; ; p = p->group_next) {
4594         q = &rdc_u_info[p->index];
4595         if (IS_STATE(q, RDC_VOL_FAILED))
4596             return (0);
4597         if (p == krdc)
4598             break;
4599     }
4600     return (1);
4601 }
4603 /*
4604  * wait here, until all in flight async i/o's have either
4605  * finished or failed. Avoid the race with r_net_state()
4606  * which tells remote end to log.
4607  */
4608 void
4609 rdc_inflwait(rdc_group_t *grp)
4610 {
4611     int bail = RDC_CLNT_TMOU * 2; /* to include retries */
4612     volatile int *inflitems;
4614     if (RDC_IS_DISKQ(grp))
4615         inflitems = (&(grp->diskq.inflitems));

```

```

4616     else
4617         inflitems = (&(grp->ra_queue.inflitems));

4619     while (*inflitems && (--bail > 0))
4620         delay(HZ);
4621 }

4623 void
4624 rdc_group_log(rdc_k_info_t *krdc, int flag, char *why)
4625 {
4626     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
4627     rdc_k_info_t *p;
4628     rdc_u_info_t *q;
4629     int do_group;
4630     int sm, um, md;
4631     disk_queue *dq;

4633     void (*flag_op)(rdc_u_info_t *urdc, int flag);

4635     ASSERT(MUTEX_HELD(&krdc->group->lock));

4637     if (!IS_ENABLED(urdc))
4638         return;

4640     rdc_many_enter(krdc);

4642     if ((flag & RDC_QUEUEING) && (!IS_STATE(urdc, RDC_SYNCING)) &&
4643         (rdc_can_queue(krdc))) {
4644         flag_op = rdc_set_flags; /* keep queuing, link error */
4645         flag &= ~RDC_FLUSH;
4646     } else {
4647         flag_op = rdc_clr_flags; /* stop queuing, user request */
4648     }

4650     do_group = 1;
4651     if (!rdc_get_vflags(urdc) & RDC_PRIMARY)
4652         do_group = 0;
4653     else if ((urdc->group_name[0] == 0) ||
4654             (rdc_get_vflags(urdc) & RDC_LOGGING) ||
4655             (rdc_get_vflags(urdc) & RDC_SYNCING))
4656         do_group = 0;
4657     if (do_group) {
4658         for (p = krdc->group_next; p != krdc; p = p->group_next) {
4659             q = &rdc_u_info[p->index];
4660             if (!IS_ENABLED(q))
4661                 continue;
4662             if ((rdc_get_vflags(q) & RDC_LOGGING) ||
4663                 (rdc_get_vflags(q) & RDC_SYNCING)) {
4664                 do_group = 0;
4665                 break;
4666             }
4667         }
4668     }
4669     if (!do_group && (flag & RDC_FORCE_GROUP))
4670         do_group = 1;

4672     rdc_many_exit(krdc);
4673     dq = &krdc->group->diskq;
4674     if (do_group) {
4675 #ifdef DEBUG
4676         cmn_err(CE_NOTE, "!SNDR:Group point-in-time for grp: %s %s:%s",
4677             urdc->group_name, urdc->primary.intf, urdc->secondary.intf);
4678 #endif
4679         DTRACE_PROBE(rdc_diskq_group_PIT);

4681         /* Set group logging at the same PIT under rdc_many_lock */

```

```

4682     rdc_many_enter(krdc);
4683     rdc_set_flags_log(urdc, RDC_LOGGING, why);
4684     if (RDC_IS_DISKQ(krdc->group))
4685         flag_op(urdc, RDC_QUEUEING);
4686     for (p = krdc->group_next; p != krdc; p = p->group_next) {
4687         q = &rdc_u_info[p->index];
4688         if (!IS_ENABLED(q))
4689             continue;
4690         rdc_set_flags_log(q, RDC_LOGGING,
4691             "consistency group member following leader");
4692         if (RDC_IS_DISKQ(p->group))
4693             flag_op(q, RDC_QUEUEING);
4694     }

4696     rdc_many_exit(krdc);

4698     /*
4699     * This can cause the async threads to fail,
4700     * which in turn will call rdc_group_log()
4701     * again. Release the lock and re-acquire.
4702     */
4703     rdc_group_exit(krdc);

4705     while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
4706         delay(2);
4707     if (!RDC_IS_DISKQ(krdc->group))
4708         RDC_ZERO_BITREF(krdc);

4710     rdc_inflwait(krdc->group);

4712     /*
4713     * a little lazy, but neat. recall dump_alloc_bufs to
4714     * ensure that the queue pointers & seq are reset properly
4715     * after we have waited for inflight stuff
4716     */
4717     while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
4718         delay(2);

4720     rdc_group_enter(krdc);
4721     if (RDC_IS_DISKQ(krdc->group) && (!flag & RDC_QUEUEING)) {
4722         /* fail or user request */
4723         RDC_ZERO_BITREF(krdc);
4724         mutex_enter(&krdc->group->diskq.disk_qlock);
4725         rdc_init_diskq_header(krdc->group,
4726             &krdc->group->diskq.disk_hdr);
4727         SET_QNXTIO(dq, QHEAD(dq));
4728         mutex_exit(&krdc->group->diskq.disk_qlock);
4729     }

4731     if (flag & RDC_ALLREMOTE) {
4732         /* Tell other node to start logging */
4733         if (krdc->lsrv && krdc->intf && !krdc->intf->if_down)
4734             (void) rdc_net_state(krdc->index,
4735                 CCIO_ENABLELOG);
4736     }

4738     if (flag & (RDC_ALLREMOTE | RDC_OTHERREMOTE)) {
4739         rdc_many_enter(krdc);
4740         for (p = krdc->group_next; p != krdc;
4741             p = p->group_next) {
4742             if (p->lsrv && krdc->intf &&
4743                 !krdc->intf->if_down) {
4744                 (void) rdc_net_state(p->index,
4745                     CCIO_ENABLELOG);
4746             }
4747         }

```

```

4748         rdc_many_exit(krdc);
4749     }
4751     rdc_write_state(urdc);
4752     for (p = krdc->group_next; p != krdc; p = p->group_next) {
4753         q = &rdc_u_info[p->index];
4754         if (!IS_ENABLED(q))
4755             continue;
4756         rdc_write_state(q);
4757     }
4758 } else {
4759     /* No point in time is possible, just deal with single set */
4761     if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
4762         halt_sync(krdc);
4763     } else {
4764         if (rdc_net_getstate(krdc, &sm, &um, &md, TRUE) < 0) {
4765             rdc_clr_flags(urdc, RDC_SYNCING);
4766             rdc_set_flags_log(urdc, RDC_LOGGING,
4767                 "failed to read remote state");
4769             rdc_write_state(urdc);
4770             while (rdc_dump_alloc_bufs_cd(krdc->index)
4771                 == EAGAIN)
4772                 delay(2);
4773             if ((RDC_IS_DISKQ(krdc->group)) &&
4774                 (!(flag & RDC_QUEUEING))) { /* fail! */
4775                 mutex_enter(QLOCK(dq));
4776                 rdc_init_diskq_header(krdc->group,
4777                     &krdc->group->diskq.disk_hdr);
4778                 SET_QNXTIO(dq, QHEAD(dq));
4779                 mutex_exit(QLOCK(dq));
4780             }
4782             return;
4783         }
4784     }
4786     if (rdc_get_vflags(urdc) & RDC_SYNCING)
4787         return;
4789     if (RDC_IS_DISKQ(krdc->group))
4790         flag_op(urdc, RDC_QUEUEING);
4792     if ((RDC_IS_DISKQ(krdc->group)) &&
4793         (!(flag & RDC_QUEUEING))) { /* fail! */
4794         RDC_ZERO_BITREF(krdc);
4795         mutex_enter(QLOCK(dq));
4796         rdc_init_diskq_header(krdc->group,
4797             &krdc->group->diskq.disk_hdr);
4798         SET_QNXTIO(dq, QHEAD(dq));
4799         mutex_exit(QLOCK(dq));
4800     }
4802     if (!(rdc_get_vflags(urdc) & RDC_LOGGING)) {
4803         rdc_set_flags_log(urdc, RDC_LOGGING, why);
4805         rdc_write_state(urdc);
4807         while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
4808             delay(2);
4809         if (!RDC_IS_DISKQ(krdc->group))
4810             RDC_ZERO_BITREF(krdc);
4812         rdc_inflwait(krdc->group);
4813         /*

```

```

4814         * a little lazy, but neat. recall dump_alloc_bufs to
4815         * ensure that the queue pointers & seq are reset
4816         * properly after we have waited for inflight stuff
4817         */
4818         while (rdc_dump_alloc_bufs_cd(krdc->index) == EAGAIN)
4819             delay(2);
4821         if (flag & RDC_ALLREMOTE) {
4822             /* Tell other node to start logging */
4823             if (krdc->lsrv && krdc->intf &&
4824                 !krdc->intf->if_down) {
4825                 (void) rdc_net_state(krdc->index,
4826                     CCIO_ENABLELOG);
4827             }
4828         }
4829     }
4830 }
4831 /*
4832 * just in case any threads were in flight during log cleanup
4833 */
4834 if (RDC_IS_DISKQ(krdc->group)) {
4835     mutex_enter(QLOCK(dq));
4836     cv_broadcast(&dq->qfullcv);
4837     mutex_exit(QLOCK(dq));
4838 }
4839 }
4841 static int
4842 _rdc_log(rdc_k_info_t *krdc, rdc_set_t *rdc_set, spcs_s_info_t kstatus)
4843 {
4844     rdc_u_info_t *urdc = &rdc_u_info[krdc->index];
4845     rdc_srv_t *svp;
4847     rdc_group_enter(krdc);
4848     if (rdc_check(krdc, rdc_set)) {
4849         rdc_group_exit(krdc);
4850         spcs_s_add(kstatus, RDC_EALREADY, rdc_set->primary.file,
4851             rdc_set->secondary.file);
4852         return (RDC_EALREADY);
4853     }
4855     svp = krdc->lsrv;
4856     if (rdc_get_vflags(urdc) & RDC_PRIMARY)
4857         krdc->intf = rdc_add_to_if(svp, &(urdc->primary.addr),
4858             &(urdc->secondary.addr), 1);
4859     else
4860         krdc->intf = rdc_add_to_if(svp, &(urdc->secondary.addr),
4861             &(urdc->primary.addr), 0);
4863     if (!krdc->intf) {
4864         rdc_group_exit(krdc);
4865         spcs_s_add(kstatus, RDC_EADDTOIF, urdc->primary.intf,
4866             urdc->secondary.intf);
4867         return (RDC_EADDTOIF);
4868     }
4870     rdc_group_log(krdc, RDC_FLUSH | RDC_ALLREMOTE, NULL);
4872     if (rdc_get_vflags(urdc) & RDC_SYNCING) {
4873         rdc_group_exit(krdc);
4874         spcs_s_add(kstatus, RDC_ESYNCING, urdc->primary.file);
4875         return (RDC_ESYNCING);
4876     }
4878     rdc_group_exit(krdc);

```



```

4880     return (0);
4881 }

4883 static int
4884 rdc_log(rdc_config_t *uparms, spcs_s_info_t kstatus)
4885 {
4886     rdc_k_info_t *krdc;
4887     int rc = 0;
4888     int index;

4890     mutex_enter(&rdc_conf_lock);
4891     index = rdc_lookup_byname(uparms->rdc_set);
4892     if (index >= 0)
4893         krdc = &rdc_k_info[index];
4894     if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
4895         mutex_exit(&rdc_conf_lock);
4896         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
4897             uparms->rdc_set->secondary.file);
4898         return (RDC_EALREADY);
4899     }

4901     set_busy(krdc);
4902     if (krdc->type_flag == 0) {
4903         /* A resume or enable failed */
4904         wakeup_busy(krdc);
4905         mutex_exit(&rdc_conf_lock);
4906         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
4907             uparms->rdc_set->secondary.file);
4908         return (RDC_EALREADY);
4909     }
4910     mutex_exit(&rdc_conf_lock);

4912     rc = _rdc_log(krdc, uparms->rdc_set, kstatus);

4914     mutex_enter(&rdc_conf_lock);
4915     wakeup_busy(krdc);
4916     mutex_exit(&rdc_conf_lock);

4918     return (rc);
4919 }

4922 static int
4923 rdc_wait(rdc_config_t *uparms, spcs_s_info_t kstatus)
4924 {
4925     rdc_k_info_t *krdc;
4926     rdc_u_info_t *urdc;
4927     int index;
4928     int need_check = 0;

4930     mutex_enter(&rdc_conf_lock);
4931     index = rdc_lookup_byname(uparms->rdc_set);
4932     if (index >= 0)
4933         krdc = &rdc_k_info[index];
4934     if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
4935         mutex_exit(&rdc_conf_lock);
4936         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
4937             uparms->rdc_set->secondary.file);
4938         return (RDC_EALREADY);
4939     }

4941     urdc = &rdc_u_info[index];
4942     if (!(rdc_get_vflags(urdc) & RDC_PRIMARY)) {
4943         mutex_exit(&rdc_conf_lock);
4944         return (0);
4945     }

```

```

4947     set_busy(krdc);
4948     if (krdc->type_flag == 0) {
4949         /* A resume or enable failed */
4950         wakeup_busy(krdc);
4951         mutex_exit(&rdc_conf_lock);
4952         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
4953             uparms->rdc_set->secondary.file);
4954         return (RDC_EALREADY);
4955     }
4956     mutex_exit(&rdc_conf_lock);

4958     rdc_group_enter(krdc);
4959     if (rdc_check(krdc, uparms->rdc_set)) {
4960         rdc_group_exit(krdc);
4961         mutex_enter(&rdc_conf_lock);
4962         wakeup_busy(krdc);
4963         mutex_exit(&rdc_conf_lock);
4964         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
4965             uparms->rdc_set->secondary.file);
4966         return (RDC_EALREADY);
4967     }

4969     if ((rdc_get_vflags(urdc) & (RDC_SYNCING | RDC_PRIMARY)) !=
4970         (RDC_SYNCING | RDC_PRIMARY)) {
4971         rdc_group_exit(krdc);
4972         mutex_enter(&rdc_conf_lock);
4973         wakeup_busy(krdc);
4974         mutex_exit(&rdc_conf_lock);
4975         return (0);
4976     }
4977     if (rdc_get_vflags(urdc) & RDC_SYNCING) {
4978         need_check = 1;
4979     }
4980     rdc_group_exit(krdc);

4982     mutex_enter(&net_blk_lock);

4984     mutex_enter(&rdc_conf_lock);
4985     wakeup_busy(krdc);
4986     mutex_exit(&rdc_conf_lock);

4988     (void) cv_wait_sig(&krdc->synccv, &net_blk_lock);

4990     mutex_exit(&net_blk_lock);
4991     if (need_check) {
4992         if (krdc->sync_done == RDC_COMPLETED) {
4993             return (0);
4994         } else if (krdc->sync_done == RDC_FAILED) {
4995             return (EIO);
4996         }
4997     }
4998     return (0);
4999 }

5002 static int
5003 rdc_health(rdc_config_t *uparms, spcs_s_info_t kstatus, int *rvp)
5004 {
5005     rdc_k_info_t *krdc;
5006     rdc_u_info_t *urdc;
5007     int rc = 0;
5008     int index;

5010     mutex_enter(&rdc_conf_lock);
5011     index = rdc_lookup_byname(uparms->rdc_set);

```

```

5012     if (index >= 0)
5013         krdc = &rdc_k_info[index];
5014     if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
5015         mutex_exit(&rdc_conf_lock);
5016         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5017                 uparms->rdc_set->secondary.file);
5018         return (RDC_EALREADY);
5019     }

5021     set_busy(krdc);
5022     if (krdc->type_flag == 0) {
5023         /* A resume or enable failed */
5024         wakeup_busy(krdc);
5025         mutex_exit(&rdc_conf_lock);
5026         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5027                 uparms->rdc_set->secondary.file);
5028         return (RDC_EALREADY);
5029     }

5031     mutex_exit(&rdc_conf_lock);

5033     rdc_group_enter(krdc);
5034     if (rdc_check(krdc, uparms->rdc_set)) {
5035         rdc_group_exit(krdc);
5036         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5037                 uparms->rdc_set->secondary.file);
5038         rc = RDC_EALREADY;
5039         goto done;
5040     }

5042     urdc = &rdc_u_info[index];
5043     if (rdc_isactive_if(&(urdc->primary.addr), &(urdc->secondary.addr)))
5044         *rvp = RDC_ACTIVE;
5045     else
5046         *rvp = RDC_INACTIVE;

5048     rdc_group_exit(krdc);

5050 done:
5051     mutex_enter(&rdc_conf_lock);
5052     wakeup_busy(krdc);
5053     mutex_exit(&rdc_conf_lock);

5055     return (rc);
5056 }

5059 static int
5060 rdc_reconfig(rdc_config_t *uparms, spcs_s_info_t kstatus)
5061 {
5062     rdc_k_info_t *krdc;
5063     rdc_u_info_t *urdc;
5064     int rc = -2;
5065     int index;

5067     mutex_enter(&rdc_conf_lock);
5068     index = rdc_lookup_byname(uparms->rdc_set);
5069     if (index >= 0)
5070         krdc = &rdc_k_info[index];
5071     if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
5072         mutex_exit(&rdc_conf_lock);
5073         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5074                 uparms->rdc_set->secondary.file);
5075         return (RDC_EALREADY);
5076     }

```

```

5078     urdc = &rdc_u_info[index];
5079     set_busy(krdc);
5080     if (krdc->type_flag == 0) {
5081         /* A resume or enable failed */
5082         wakeup_busy(krdc);
5083         mutex_exit(&rdc_conf_lock);
5084         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5085                 uparms->rdc_set->secondary.file);
5086         return (RDC_EALREADY);
5087     }

5089     mutex_exit(&rdc_conf_lock);

5091     rdc_group_enter(krdc);
5092     if (rdc_check(krdc, uparms->rdc_set)) {
5093         rdc_group_exit(krdc);
5094         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5095                 uparms->rdc_set->secondary.file);
5096         rc = RDC_EALREADY;
5097         goto done;
5098     }
5099     if ((rdc_get_vflags(urdc) & RDC_BMP_FAILED) && (krdc->bitmapfd))
5100         (void) rdc_reset_bitmap(krdc);

5102     /* Move to a new bitmap if necessary */
5103     if (strcmp(urdc->primary.bitmap, uparms->rdc_set->primary.bitmap,
5104             NSC_MAXPATH) != 0) {
5105         if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
5106             rc = rdc_move_bitmap(krdc,
5107                 uparms->rdc_set->primary.bitmap);
5108         } else {
5109             (void) strncpy(urdc->primary.bitmap,
5110                 uparms->rdc_set->primary.bitmap, NSC_MAXPATH);
5111             /* simulate a succesful rdc_move_bitmap */
5112             rc = 0;
5113         }
5114     }
5115     if (strcmp(urdc->secondary.bitmap, uparms->rdc_set->secondary.bitmap,
5116             NSC_MAXPATH) != 0) {
5117         if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
5118             (void) strncpy(urdc->secondary.bitmap,
5119                 uparms->rdc_set->secondary.bitmap, NSC_MAXPATH);
5120             /* simulate a succesful rdc_move_bitmap */
5121             rc = 0;
5122         } else {
5123             rc = rdc_move_bitmap(krdc,
5124                 uparms->rdc_set->secondary.bitmap);
5125         }
5126     }
5127     if (rc == -1) {
5128         rdc_group_exit(krdc);
5129         spcs_s_add(kstatus, RDC_EBMPRECONFIG,
5130                 uparms->rdc_set->secondary.intf,
5131                 uparms->rdc_set->secondary.file);
5132         rc = RDC_EBMPRECONFIG;
5133         goto done;
5134     }

5136     /*
5137     * At this point we fail any other type of reconfig
5138     * if not in logging mode and we did not do a bitmap reconfig
5139     */

5141     if (!(rdc_get_vflags(urdc) & RDC_LOGGING) && rc == -2) {
5142         /* no other changes possible unless logging */
5143         rdc_group_exit(krdc);

```

```

5144         spcs_s_add(kstatus, RDC_ENOTLOGGING,
5145                 uparms->rdc_set->primary.intf,
5146                 uparms->rdc_set->primary.file,
5147                 uparms->rdc_set->secondary.intf,
5148                 uparms->rdc_set->secondary.file);
5149         rc = RDC_ENOTLOGGING;
5150         goto done;
5151     }
5152     rc = 0;
5153     /* Change direct file if necessary */
5154     if ((rdc_get_vflags(urdc) & RDC_PRIMARY) &&
5155         strcmp(urdc->direct_file, uparms->rdc_set->direct_file,
5156             NSC_MAXPATH)) {
5157         if (!(rdc_get_vflags(urdc) & RDC_LOGGING)) {
5158             rdc_group_exit(krdc);
5159             goto notlogging;
5160         }
5161         rdc_close_direct(krdc);
5162         (void) strncpy(urdc->direct_file, uparms->rdc_set->direct_file,
5163             NSC_MAXPATH);
5164     }
5165     if (urdc->direct_file[0]) {
5166         if (rdc_open_direct(krdc) == NULL)
5167             rdc_set_flags(urdc, RDC_FCAL_FAILED);
5168         else
5169             rdc_clr_flags(urdc, RDC_FCAL_FAILED);
5170     }
5171 }
5172
5173 rdc_group_exit(krdc);
5174
5175 /* Change group if necessary */
5176 if (strcmp(urdc->group_name, uparms->rdc_set->group_name,
5177     NSC_MAXPATH) != 0) {
5178     char orig_group[NSC_MAXPATH];
5179     if (!(rdc_get_vflags(urdc) & RDC_LOGGING))
5180         goto notlogging;
5181     mutex_enter(&rdc_conf_lock);
5182
5183     (void) strncpy(orig_group, urdc->group_name, NSC_MAXPATH);
5184     (void) strncpy(urdc->group_name, uparms->rdc_set->group_name,
5185         NSC_MAXPATH);
5186
5187     rc = change_group(krdc, uparms->options);
5188     if (rc == RDC_EQNOADD) {
5189         mutex_exit(&rdc_conf_lock);
5190         spcs_s_add(kstatus, RDC_EQNOADD,
5191             uparms->rdc_set->disk_queue);
5192         goto done;
5193     } else if (rc < 0) {
5194         (void) strncpy(urdc->group_name, orig_group,
5195             NSC_MAXPATH);
5196         mutex_exit(&rdc_conf_lock);
5197         spcs_s_add(kstatus, RDC_EGROUP,
5198             urdc->primary.intf, urdc->primary.file,
5199             urdc->secondary.intf, urdc->secondary.file,
5200             uparms->rdc_set->group_name);
5201         rc = RDC_EGROUP;
5202         goto done;
5203     }
5204
5205     mutex_exit(&rdc_conf_lock);
5206
5207     if (rc >= 0) {
5208         if (!(rdc_get_vflags(urdc) & RDC_LOGGING))
5209             goto notlogging;

```

```

5210         if (uparms->options & RDC_OPT_ASYNC) {
5211             mutex_enter(&rdc_conf_lock);
5212             krdc->type_flag |= RDC_ASYNCMODE;
5213             mutex_exit(&rdc_conf_lock);
5214             if (uparms->options & RDC_OPT_PRIMARY)
5215                 krdc->bitmap_ref =
5216                     (uchar_t *)kmem_zalloc(
5217                         (krdc->bitmap_size * BITS_IN_BYTE *
5218                             BMAP_REF_PREF_SIZE), KM_SLEEP);
5219             rdc_group_enter(krdc);
5220             rdc_set_flags(urdc, RDC_ASYNC);
5221             rdc_group_exit(krdc);
5222         } else {
5223             mutex_enter(&rdc_conf_lock);
5224             krdc->type_flag &= ~RDC_ASYNCMODE;
5225             mutex_exit(&rdc_conf_lock);
5226             rdc_group_enter(krdc);
5227             rdc_clr_flags(urdc, RDC_ASYNC);
5228             rdc_group_exit(krdc);
5229             if (krdc->bitmap_ref) {
5230                 kmem_free(krdc->bitmap_ref,
5231                     (krdc->bitmap_size * BITS_IN_BYTE *
5232                         BMAP_REF_PREF_SIZE));
5233                 krdc->bitmap_ref = NULL;
5234             }
5235         }
5236     }
5237 } else {
5238     if (((uparms->options & RDC_OPT_ASYNC) == 0) &&
5239         ((krdc->type_flag & RDC_ASYNCMODE) != 0) ||
5240         ((uparms->options & RDC_OPT_ASYNC) != 0) &&
5241         ((krdc->type_flag & RDC_ASYNCMODE) == 0)) {
5242         if (!(rdc_get_vflags(urdc) & RDC_LOGGING))
5243             goto notlogging;
5244
5245         if (krdc->group->count > 1) {
5246             spcs_s_add(kstatus, RDC_EGROUPMODE);
5247             rc = RDC_EGROUPMODE;
5248             goto done;
5249         }
5250     }
5251
5252     /* Switch sync/async if necessary */
5253     if (krdc->group->count == 1) {
5254         /* Only member of group. Can change sync/async */
5255         if (((uparms->options & RDC_OPT_ASYNC) == 0) &&
5256             ((krdc->type_flag & RDC_ASYNCMODE) != 0)) {
5257             if (!(rdc_get_vflags(urdc) & RDC_LOGGING))
5258                 goto notlogging;
5259             /* switch to sync */
5260             mutex_enter(&rdc_conf_lock);
5261             krdc->type_flag &= ~RDC_ASYNCMODE;
5262             if (RDC_IS_DISKQ(krdc->group)) {
5263                 krdc->group->flags &= ~RDC_DISKQUE;
5264                 krdc->group->flags |= RDC_MEMQUE;
5265                 rdc_unintercept_diskq(krdc->group);
5266                 mutex_enter(&krdc->group->diskqmutex);
5267                 rdc_close_diskq(krdc->group);
5268                 mutex_exit(&krdc->group->diskqmutex);
5269                 bzero(&urdc->disk_queue,
5270                     sizeof(urdc->disk_queue));
5271             }
5272             mutex_exit(&rdc_conf_lock);
5273             rdc_group_enter(krdc);
5274             rdc_clr_flags(urdc, RDC_ASYNC);
5275             rdc_group_exit(krdc);

```

```

5276     if (krdc->bitmap_ref) {
5277         kmem_free(krdc->bitmap_ref,
5278                 (krdc->bitmap_size * BITS_IN_BYTE *
5279                  BMAP_REF_PREF_SIZE));
5280         krdc->bitmap_ref = NULL;
5281     }
5282     } else if (((uparms->options & RDC_OPT_ASYNC) != 0) &&
5283               ((krdc->type_flag & RDC_ASYNCMODE) == 0)) {
5284         if (!(rdc_get_vflags(urdc) & RDC_LOGGING))
5285             goto notlogging;
5286         /* switch to async */
5287         mutex_enter(&rdc_conf_lock);
5288         krdc->type_flag |= RDC_ASYNCMODE;
5289         mutex_exit(&rdc_conf_lock);
5290         if (uparms->options & RDC_OPT_PRIMARY)
5291             krdc->bitmap_ref =
5292                 (uchar_t *)kmem_zalloc(
5293                     (krdc->bitmap_size * BITS_IN_BYTE *
5294                      BMAP_REF_PREF_SIZE), KM_SLEEP);
5295         rdc_group_enter(krdc);
5296         rdc_set_flags(urdc, RDC_ASYNC);
5297         rdc_group_exit(krdc);
5298     }
5299 }
5300
5301 /* Reverse concept of primary and secondary */
5302 if ((uparms->options & RDC_OPT_REVERSE_ROLE) != 0) {
5303     rdc_set_t rdc_set;
5304     struct netbuf paddr, saddr;
5305
5306     mutex_enter(&rdc_conf_lock);
5307
5308     /*
5309      * Disallow role reversal for advanced configurations
5310      */
5311     if (IS_MANY(krdc) || IS_MULTI(krdc)) {
5312         mutex_exit(&rdc_conf_lock);
5313         spcs_s_add(kstatus, RDC_EMASTER, urdc->primary.intf,
5314                 urdc->primary.file, urdc->secondary.intf,
5315                 urdc->secondary.file);
5316         return (RDC_EMASTER);
5317     }
5318     bzero((void *) &rdc_set, sizeof (rdc_set_t));
5319     dup_rdc_netbuf(&urdc->primary.addr, &saddr);
5320     dup_rdc_netbuf(&urdc->secondary.addr, &paddr);
5321     free_rdc_netbuf(&urdc->primary.addr);
5322     free_rdc_netbuf(&urdc->secondary.addr);
5323     dup_rdc_netbuf(&saddr, &urdc->secondary.addr);
5324     dup_rdc_netbuf(&paddr, &urdc->primary.addr);
5325     free_rdc_netbuf(&paddr);
5326     free_rdc_netbuf(&saddr);
5327     /* copy primary parts of urdc to rdc_set field by field */
5328     (void) strncpy(rdc_set.primary.intf, urdc->primary.intf,
5329                 MAX_RDC_HOST_SIZE);
5330     (void) strncpy(rdc_set.primary.file, urdc->primary.file,
5331                 NSC_MAXPATH);
5332     (void) strncpy(rdc_set.primary.bitmap, urdc->primary.bitmap,
5333                 NSC_MAXPATH);
5334
5335     /* Now overwrite urdc primary */
5336     (void) strncpy(urdc->primary.intf, urdc->secondary.intf,
5337                 MAX_RDC_HOST_SIZE);
5338     (void) strncpy(urdc->primary.file, urdc->secondary.file,
5339                 NSC_MAXPATH);
5340     (void) strncpy(urdc->primary.bitmap, urdc->secondary.bitmap,

```

```

5342         NSC_MAXPATH);
5343
5344     /* Now overwrite urdc secondary */
5345     (void) strncpy(urdc->secondary.intf, rdc_set.primary.intf,
5346                 MAX_RDC_HOST_SIZE);
5347     (void) strncpy(urdc->secondary.file, rdc_set.primary.file,
5348                 NSC_MAXPATH);
5349     (void) strncpy(urdc->secondary.bitmap, rdc_set.primary.bitmap,
5350                 NSC_MAXPATH);
5351
5352     if (rdc_get_vflags(urdc) & RDC_PRIMARY) {
5353         rdc_clr_flags(urdc, RDC_PRIMARY);
5354         if (krdc->intf) {
5355             krdc->intf->issecondary = 1;
5356             krdc->intf->isprimary = 0;
5357             krdc->intf->if_down = 1;
5358         }
5359     } else {
5360         rdc_set_flags(urdc, RDC_PRIMARY);
5361         if (krdc->intf) {
5362             krdc->intf->issecondary = 0;
5363             krdc->intf->isprimary = 1;
5364             krdc->intf->if_down = 1;
5365         }
5366     }
5367
5368     if ((rdc_get_vflags(urdc) & RDC_PRIMARY) &&
5369         ((krdc->type_flag & RDC_ASYNCMODE) != 0)) {
5370         if (!krdc->bitmap_ref)
5371             krdc->bitmap_ref =
5372                 (uchar_t *)kmem_zalloc((krdc->bitmap_size *
5373                     BITS_IN_BYTE * BMAP_REF_PREF_SIZE),
5374                     KM_SLEEP);
5375         if (krdc->bitmap_ref == NULL) {
5376             cmn_err(CE_WARN,
5377                 "!rdc_reconfig: bitmap_ref alloc %"
5378                 NSC_SZFMT " failed",
5379                 krdc->bitmap_size * BITS_IN_BYTE *
5380                 BMAP_REF_PREF_SIZE);
5381             mutex_exit(&rdc_conf_lock);
5382             return (-1);
5383         }
5384     }
5385
5386     if ((rdc_get_vflags(urdc) & RDC_PRIMARY) &&
5387         (rdc_get_vflags(urdc) & RDC_SYNC_NEEDED)) {
5388         /* Primary, so reverse sync needed */
5389         rdc_many_enter(krdc);
5390         rdc_clr_flags(urdc, RDC_SYNC_NEEDED);
5391         rdc_set_mflags(urdc, RDC_RSYNC_NEEDED);
5392         rdc_many_exit(krdc);
5393     } else if (rdc_get_vflags(urdc) & RDC_RSYNC_NEEDED) {
5394         /* Secondary, so forward sync needed */
5395         rdc_many_enter(krdc);
5396         rdc_clr_flags(urdc, RDC_RSYNC_NEEDED);
5397         rdc_set_flags(urdc, RDC_SYNC_NEEDED);
5398         rdc_many_exit(krdc);
5399     }
5400
5401     /*
5402      * rewrite bitmap header
5403      */
5404     rdc_write_state(urdc);
5405     mutex_exit(&rdc_conf_lock);
5406 }

```

```

5408 done:
5409     mutex_enter(&rdc_conf_lock);
5410     wakeup_busy(krdc);
5411     mutex_exit(&rdc_conf_lock);
5413     return (rc);
5415 notlogging:
5416     /* no other changes possible unless logging */
5417     mutex_enter(&rdc_conf_lock);
5418     wakeup_busy(krdc);
5419     mutex_exit(&rdc_conf_lock);
5420     spcs_s_add(kstatus, RDC_ENOTLOGGING, urdc->primary.intf,
5421             urdc->primary.file, urdc->secondary.intf,
5422             urdc->secondary.file);
5423     return (RDC_ENOTLOGGING);
5424 }
5426 static int
5427 rdc_reset(rdc_config_t *uparms, spcs_s_info_t kstatus)
5428 {
5429     rdc_k_info_t *krdc;
5430     rdc_u_info_t *urdc;
5431     int rc = 0;
5432     int index;
5433     int cleared_error = 0;
5435     mutex_enter(&rdc_conf_lock);
5436     index = rdc_lookup_byname(uparms->rdc_set);
5437     if (index >= 0)
5438         krdc = &rdc_k_info[index];
5439     if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
5440         mutex_exit(&rdc_conf_lock);
5441         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5442             uparms->rdc_set->secondary.file);
5443         return (RDC_EALREADY);
5444     }
5446     urdc = &rdc_u_info[index];
5447     set_busy(krdc);
5448     if (krdc->type_flag == 0) {
5449         /* A resume or enable failed */
5450         wakeup_busy(krdc);
5451         mutex_exit(&rdc_conf_lock);
5452         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5453             uparms->rdc_set->secondary.file);
5454         return (RDC_EALREADY);
5455     }
5457     mutex_exit(&rdc_conf_lock);
5459     rdc_group_enter(krdc);
5460     if (rdc_check(krdc, uparms->rdc_set)) {
5461         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5462             uparms->rdc_set->secondary.file);
5463         rc = RDC_EALREADY;
5464         goto done;
5465     }
5467     if ((rdc_get_vflags(urdc) & RDC_BMP_FAILED) && (krdc->bitmapfd)) {
5468         if (rdc_reset_bitmap(krdc) == 0)
5469             cleared_error++;
5470     }
5472     /* Fix direct file if necessary */
5473     if ((rdc_get_vflags(urdc) & RDC_PRIMARY) && urdc->direct_file[0]) {

```

```

5474         if (rdc_open_direct(krdc) == NULL)
5475             rdc_set_flags(urdc, RDC_FCAL_FAILED);
5476         else {
5477             rdc_clr_flags(urdc, RDC_FCAL_FAILED);
5478             cleared_error++;
5479         }
5480     }
5482     if ((rdc_get_vflags(urdc) & RDC_VOL_FAILED)) {
5483         rdc_many_enter(krdc);
5484         rdc_clr_flags(urdc, RDC_VOL_FAILED);
5485         cleared_error++;
5486         rdc_many_exit(krdc);
5487     }
5489     if (cleared_error) {
5490         /* Cleared an error so we should be in logging mode */
5491         rdc_set_flags_log(urdc, RDC_LOGGING, "set reset");
5492     }
5493     rdc_group_exit(krdc);
5495     if ((rdc_get_vflags(urdc) & RDC_DISKQ_FAILED))
5496         rdc_unfail_diskq(krdc);
5498 done:
5499     mutex_enter(&rdc_conf_lock);
5500     wakeup_busy(krdc);
5501     mutex_exit(&rdc_conf_lock);
5503     return (rc);
5504 }
5507 static int
5508 rdc_tunable(rdc_config_t *uparms, spcs_s_info_t kstatus)
5509 {
5510     rdc_k_info_t *krdc;
5511     rdc_u_info_t *urdc;
5512     rdc_k_info_t *p;
5513     rdc_u_info_t *q;
5514     int rc = 0;
5515     int index;
5517     mutex_enter(&rdc_conf_lock);
5518     index = rdc_lookup_byname(uparms->rdc_set);
5519     if (index >= 0)
5520         krdc = &rdc_k_info[index];
5521     if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
5522         mutex_exit(&rdc_conf_lock);
5523         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5524             uparms->rdc_set->secondary.file);
5525         return (RDC_EALREADY);
5526     }
5528     urdc = &rdc_u_info[index];
5529     set_busy(krdc);
5530     if (krdc->type_flag == 0) {
5531         /* A resume or enable failed */
5532         wakeup_busy(krdc);
5533         mutex_exit(&rdc_conf_lock);
5534         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5535             uparms->rdc_set->secondary.file);
5536         return (RDC_EALREADY);
5537     }
5539     mutex_exit(&rdc_conf_lock);

```

```

5541 rdc_group_enter(krdc);
5542 if (rdc_check(krdc, uparms->rdc_set)) {
5543     spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5544             uparms->rdc_set->secondary.file);
5545     rc = RDC_EALREADY;
5546     goto done;
5547 }
5548
5549 if (uparms->rdc_set->maxqfbas > 0) {
5550     urdc->maxqfbas = uparms->rdc_set->maxqfbas;
5551     rdc_write_state(urdc);
5552     for (p = krdc->group_next; p != krdc; p = p->group_next) {
5553         q = &rdc_u_info[p->index];
5554         q->maxqfbas = urdc->maxqfbas;
5555         rdc_write_state(q);
5556     }
5557 }
5558
5559 if (uparms->rdc_set->maxqitems > 0) {
5560     urdc->maxqitems = uparms->rdc_set->maxqitems;
5561     rdc_write_state(urdc);
5562     for (p = krdc->group_next; p != krdc; p = p->group_next) {
5563         q = &rdc_u_info[p->index];
5564         q->maxqitems = urdc->maxqitems;
5565         rdc_write_state(q);
5566     }
5567 }
5568
5569 if (uparms->options & RDC_OPT_SET_QNOBLOCK) {
5570     disk_queue *que;
5571
5572     if (!RDC_IS_DISKQ(krdc->group)) {
5573         spcs_s_add(kstatus, RDC_EQNOQUEUE, urdc->primary.intf,
5574             urdc->primary.file, urdc->secondary.intf,
5575             urdc->secondary.file);
5576         rc = RDC_EQNOQUEUE;
5577         goto done;
5578     }
5579
5580     que = &krdc->group->diskq;
5581     mutex_enter(QLOCK(que));
5582     SET_QSTATE(que, RDC_QNOBLOCK);
5583     /* queue will fail if this fails */
5584     (void) rdc_stamp_diskq(krdc, 0, RDC_GROUP_LOCKED);
5585     mutex_exit(QLOCK(que));
5586 }
5587
5588 if (uparms->options & RDC_OPT_CLR_QNOBLOCK) {
5589     disk_queue *que;
5590
5591     if (!RDC_IS_DISKQ(krdc->group)) {
5592         spcs_s_add(kstatus, RDC_EQNOQUEUE, urdc->primary.intf,
5593             urdc->primary.file, urdc->secondary.intf,
5594             urdc->secondary.file);
5595         rc = RDC_EQNOQUEUE;
5596         goto done;
5597     }
5598     que = &krdc->group->diskq;
5599     mutex_enter(QLOCK(que));
5600     CLR_QSTATE(que, RDC_QNOBLOCK);
5601     /* queue will fail if this fails */
5602     (void) rdc_stamp_diskq(krdc, 0, RDC_GROUP_LOCKED);
5603     mutex_exit(QLOCK(que));
5604 }

```

```

5606     }
5607     if (uparms->rdc_set->asynchtr > 0) {
5608         urdc->asynchtr = uparms->rdc_set->asynchtr;
5609         rdc_write_state(urdc);
5610         for (p = krdc->group_next; p != krdc; p = p->group_next) {
5611             q = &rdc_u_info[p->index];
5612             q->asynchtr = urdc->asynchtr;
5613             rdc_write_state(q);
5614         }
5615     }
5616
5617     if (uparms->rdc_set->autosync >= 0) {
5618         if (uparms->rdc_set->autosync == 0)
5619             urdc->autosync = 0;
5620         else
5621             urdc->autosync = 1;
5622     }
5623     rdc_write_state(urdc);
5624
5625     /* Changed autosync, so update rest of the group */
5626
5627     for (p = krdc->group_next; p != krdc; p = p->group_next) {
5628         q = &rdc_u_info[p->index];
5629         q->autosync = urdc->autosync;
5630         rdc_write_state(q);
5631     }
5632 }
5633
5634 done:
5635     rdc_group_exit(krdc);
5636
5637     mutex_enter(&rdc_conf_lock);
5638     wakeup_busy(krdc);
5639     mutex_exit(&rdc_conf_lock);
5640
5641     return (rc);
5642 }
5643
5644 static int
5645 rdc_status(void *arg, int mode, rdc_config_t *uparms, spcs_s_info_t kstatus)
5646 {
5647     rdc_k_info_t *krdc;
5648     rdc_u_info_t *urdc;
5649     disk_queue *dq;
5650     int rc = 0;
5651     int index;
5652     char *ptr;
5653     extern int rdc_status_copy32(const void *, void *, size_t, int);
5654
5655     mutex_enter(&rdc_conf_lock);
5656     index = rdc_lookup_byname(uparms->rdc_set);
5657     if (index >= 0)
5658         krdc = &rdc_k_info[index];
5659     if (index < 0 || (krdc->type_flag & RDC_DISABLEPEND)) {
5660         mutex_exit(&rdc_conf_lock);
5661         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5662             uparms->rdc_set->secondary.file);
5663         return (RDC_EALREADY);
5664     }

```

```

5666     set_busy(krdc);
5667     if (krdc->type_flag == 0) {
5668         /* A resume or enable failed */
5669         wakeup_busy(krdc);
5670         mutex_exit(&krdc_conf_lock);
5671         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5672                 uparms->rdc_set->secondary.file);
5673         return (RDC_EALREADY);
5674     }
5676     mutex_exit(&krdc_conf_lock);
5678     rdc_group_enter(krdc);
5679     if (rdc_check(krdc, uparms->rdc_set)) {
5680         rdc_group_exit(krdc);
5681         spcs_s_add(kstatus, RDC_EALREADY, uparms->rdc_set->primary.file,
5682                 uparms->rdc_set->secondary.file);
5683         rc = RDC_EALREADY;
5684         goto done;
5685     }
5687     urdc = &rdc_u_info[index];
5689     /*
5690     * sneak out qstate in urdc->flags
5691     * this is harmless because it's value is not used
5692     * in urdc->flags. the real qstate is kept in
5693     * group->diskq->disk_hdr.h.state
5694     */
5695     if (RDC_IS_DISKQ(krdc->group)) {
5696         dqp = &krdc->group->diskq;
5697         if (IS_QSTATE(dqp, RDC_QNOBLOCK))
5698             urdc->flags |= RDC_QNOBLOCK;
5699     }
5701     if (ddi_model_convert_from(mode & FMODELS) == DDI_MODEL_ILP32) {
5702         ptr = (char *)arg + offsetof(struct rdc_config32, rdc_set);
5703         rc = rdc_status_copy32(urdc, ptr, sizeof (struct rdc_set32),
5704                               mode);
5705     } else {
5706         ptr = (char *)arg + offsetof(struct rdc_config, rdc_set);
5707         rc = ddi_copyout(urdc, ptr, sizeof (struct rdc_set), mode);
5708     }
5709     /* clear out qstate from flags */
5710     urdc->flags &= ~RDC_QNOBLOCK;
5712     if (rc)
5713         rc = EFAULT;
5715     rdc_group_exit(krdc);
5716 done:
5717     mutex_enter(&krdc_conf_lock);
5718     wakeup_busy(krdc);
5719     mutex_exit(&krdc_conf_lock);
5721     return (rc);
5722 }

```

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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 *
25 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
26 */

28 /*
29 * Storage Volume Character and Block Driver (SV)
30 *
31 * This driver implements a simplistic /dev/{r}dsk/ interface to a
32 * specified disk volume that is otherwise managed by the Prism
33 * software. The SV driver layers itself onto the underlying disk
34 * device driver by changing function pointers in the cb_ops
35 * structure.
36 *
37 * CONFIGURATION:
38 *
39 * 1. Configure the driver using the svadm utility.
40 * 2. Access the device as before through /dev/rdsk/c?t?d?s?
41 *
42 * LIMITATIONS:
43 *
44 * This driver should NOT be used to share a device between another
45 * DataServices user interface module (e.g., STE) and a user accessing
46 * the device through the block device in O_WRITE mode. This is because
47 * writes through the block device are asynchronous (due to the page
48 * cache) and so consistency between the block device user and the
49 * STE user cannot be guaranteed.
50 *
51 * Data is copied between system struct buf(9s) and nsc_vec_t. This is
52 * wasteful and slow.
53 */

55 #include <sys/debug.h>
56 #include <sys/types.h>

58 #include <sys/ksynch.h>
59 #include <sys/kmem.h>
60 #include <sys/errno.h>
61 #include <sys/varargs.h>

```

```

62 #include <sys/file.h>
63 #include <sys/open.h>
64 #include <sys/conf.h>
65 #include <sys/cred.h>
66 #include <sys/buf.h>
67 #include <sys/uio.h>
68 #ifndef DS_DDICT
69 #include <sys/pathname.h>
70 #endif
71 #include <sys/aio_req.h>
72 #include <sys/dkio.h>
73 #include <sys/vtoc.h>
74 #include <sys/cmn_err.h>
75 #include <sys/modctl.h>
76 #include <sys/ddi.h>
77 #include <sys/sysmacros.h>
78 #endif /* !codereview */
79 #include <sys/sunddi.h>
80 #include <sys/sunldi.h>
81 #include <sys/nsctl/nsvers.h>

83 #include <sys/nsc_thread.h>
84 #include <sys/unistat/spcs_s.h>
85 #include <sys/unistat/spcs_s_k.h>
86 #include <sys/unistat/spcs_errors.h>

88 #ifdef DS_DDICT
89 #include "../contract.h"
90 #endif

92 #include "../nsctl.h"

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

97 #include "sv.h"
98 #include "sv_impl.h"
99 #include "sv_efi.h"

101 #define MAX_EINTR_COUNT 1000

103 /*
104  * sv_mod_status
105  */
106 #define SV_PREVENT_UNLOAD 1
107 #define SV_ALLOW_UNLOAD 2

109 static const int sv_major_rev = ISS_VERSION_MAJ;          /* Major number */
110 static const int sv_minor_rev = ISS_VERSION_MIN;          /* Minor number */
111 static const int sv_micro_rev = ISS_VERSION_MIC;          /* Micro number */
112 static const int sv_baseline_rev = ISS_VERSION_NUM;       /* Baseline number */

114 #ifndef DKIOCPARTITION
115 /*
116  * CRC32 polynomial table needed for computing the checksums
117  * in an EFI vtoc.
118  */
119 static const uint32_t sv_crc32_table[256] = { CRC32_TABLE };
120 #endif

122 static clock_t sv_config_time;          /* Time of successful {en,dis}able */
123 static int sv_debug;                    /* Set non-zero for debug to syslog */
124 static int sv_mod_status;               /* Set to prevent modunload */

126 static dev_info_t *sv_dip;              /* Single DIP for driver */
127 static kmutex_t sv_mutex;               /* Protect global lists, etc. */

```



```

129 static nsc_mem_t      *sv_mem;      /* nsctl memory allocator token */

132 /*
133  * Per device and per major state.
134  */

136 #ifndef _SunOS_5_6
137 #define UNSAFE_ENTER()
138 #define UNSAFE_EXIT()
139 #else
140 #define UNSAFE_ENTER()  mutex_enter(&unsafe_driver)
141 #define UNSAFE_EXIT()  mutex_exit(&unsafe_driver)
142 #endif

144 /* hash table of major dev structures */
145 static sv_maj_t *sv_majors[SV_MAJOR_HASH_CNT] = {0};
146 static sv_dev_t *sv_devs;           /* array of per device structures */
147 static int sv_max_devices;          /* SV version of nsc_max_devices() */
148 static int sv_ndevices;             /* number of SV enabled devices */

150 /*
151  * Threading.
152  */

154 int sv_threads_max = 1024;          /* maximum # to dynamically alloc */
155 int sv_threads = 32;                /* # to pre-allocate (see sv.conf) */
156 int sv_threads_extra = 0;           /* addl # we would have alloc'ed */

158 static nstset_t *sv_tset;           /* the threadset pointer */

160 static int sv_threads_hysteresis = 4; /* hysteresis for threadset resizing */
161 static int sv_threads_dev = 2;       /* # of threads to alloc per device */
162 static int sv_threads_inc = 8;       /* increment for changing the set */
163 static int sv_threads_needed;        /* number of threads needed */
164 static int sv_no_threads;            /* number of nsc_create errors */
165 static int sv_max_nlive;             /* max number of threads running */

169 /*
170  * nsctl fd callbacks.
171  */

173 static int svattach_fd(blind_t);
174 static int svdetach_fd(blind_t);

176 static nsc_def_t sv_fd_def[] = {
177     { "Attach",      (uintptr_t)svattach_fd, },
178     { "Detach",      (uintptr_t)svdetach_fd, },
179     { 0, 0, }
180 };

182 /*
183  * cb_ops functions.
184  */

186 static int svopen(dev_t *, int, int, cred_t *);
187 static int svclose(dev_t, int, int, cred_t *);
188 static int svioctl(dev_t, int, intptr_t, int, cred_t *, int *);
189 static int svprint(dev_t, char *);

191 /*
192  * These next functions are layered into the underlying driver's devops.
193  */

```

```

195 static int sv_lyr_open(dev_t *, int, int, cred_t *);
196 static int sv_lyr_close(dev_t, int, int, cred_t *);
197 static int sv_lyr_strategy(struct buf *);
198 static int sv_lyr_read(dev_t, struct uio *, cred_t *);
199 static int sv_lyr_write(dev_t, struct uio *, cred_t *);
200 static int sv_lyr_aread(dev_t, struct aio_req *, cred_t *);
201 static int sv_lyr_awrite(dev_t, struct aio_req *, cred_t *);
202 static int sv_lyr_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

204 static struct cb_ops sv_cb_ops = {
205     svopen,          /* open */
206     svclose,         /* close */
207     nulldev,         /* strategy */
208     svprint,
209     nodev,           /* dump */
210     nodev,           /* read */
211     nodev,           /* write */
212     svioctl,
213     nodev,           /* devmap */
214     nodev,           /* mmap */
215     nodev,           /* segmap */
216     nochpoll,       /* poll */
217     ddi_prop_op,    /* NOT a stream */
218     NULL,
219     D_NEW | D_MP | D_64BIT,
220     CB_REV,
221     nodev,           /* aread */
222     nodev,           /* awrite */
223 };

226 /*
227  * dev_ops functions.
228  */

230 static int sv_getinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
231 static int sv_attach(dev_info_t *, ddi_attach_cmd_t);
232 static int sv_detach(dev_info_t *, ddi_detach_cmd_t);

234 static struct dev_ops sv_ops = {
235     DEVO_REV,
236     0,
237     sv_getinfo,
238     nulldev,         /* identify */
239     nulldev,         /* probe */
240     sv_attach,
241     sv_detach,
242     nodev,           /* reset */
243     &sv_cb_ops,
244     (struct bus_ops *)0
245 };

247 /*
248  * Module linkage.
249  */

251 extern struct mod_ops mod_driverops;

253 static struct modldrv modldrv = {
254     &mod_driverops,
255     "nws:Storage Volume:" ISS_VERSION_STR,
256     &sv_ops
257 };

259 static struct modlinkage modlinkage = {

```

```

260     MODREV_1,
261     &modldrv,
262     0
263 };

266 int
267 _init(void)
268 {
269     int error;

271     mutex_init(&sv_mutex, NULL, MUTEX_DRIVER, NULL);

273     if ((error = mod_install(&modlinkage)) != 0) {
274         mutex_destroy(&sv_mutex);
275         return (error);
276     }

278 #ifdef DEBUG
279     cmn_err(CE_CONT, "!sv (revision %d.%d.%d.%d, %s, %s)\n",
280            sv_major_rev, sv_minor_rev, sv_micro_rev, sv_baseline_rev,
281            ISS_VERSION_STR, BUILD_DATE_STR);
282 #else
283     if (sv_micro_rev) {
284         cmn_err(CE_CONT, "!sv (revision %d.%d.%d, %s, %s)\n",
285            sv_major_rev, sv_minor_rev, sv_micro_rev,
286            ISS_VERSION_STR, BUILD_DATE_STR);
287     } else {
288         cmn_err(CE_CONT, "!sv (revision %d.%d, %s, %s)\n",
289            sv_major_rev, sv_minor_rev,
290            ISS_VERSION_STR, BUILD_DATE_STR);
291     }
292 #endif

294     return (error);
295 }

298 int
299 _fini(void)
300 {
301     int error;

303     if ((error = mod_remove(&modlinkage)) != 0)
304         return (error);

306     mutex_destroy(&sv_mutex);

308     return (error);
309 }

312 int
313 _info(struct modinfo *modinfop)
314 {
315     return (mod_info(&modlinkage, modinfop));
316 }

319 /*
320  * Locking & State.
321  *
322  * sv_mutex protects config information - sv_maj_t and sv_dev_t lists;
323  * threadset creation and sizing; sv_ndevices.
324  *
325  * If we need to hold both sv_mutex and sv_lock, then the sv_mutex

```

```

326  * must be acquired first.
327  *
328  * sv_lock protects the sv_dev_t structure for an individual device.
329  *
330  * sv_olock protects the otyp/open members of the sv_dev_t.  If we need
331  * to hold both sv_lock and sv_olock, then the sv_lock must be acquired
332  * first.
333  *
334  * nsc_reserve/nsc_release are used in NSC_MULTI mode to allow multiple
335  * I/O operations to a device simultaneously, as above.
336  *
337  * All nsc_open/nsc_close/nsc_reserve/nsc_release operations that occur
338  * with sv_lock write-locked must be done with (sv_state == SV_PENDING)
339  * and (sv_pending == curthread) so that any recursion through
340  * sv_lyr_open/sv_lyr_close can be detected.
341  */

344 static int
345 sv_init_devs(void)
346 {
347     int i;

349     ASSERT(MUTEX_HELD(&sv_mutex));

351     if (sv_max_devices > 0)
352         return (0);

354     sv_max_devices = nsc_max_devices();

356     if (sv_max_devices <= 0) {
357         /* nsctl is not attached (nskernd not running) */
358         if (sv_debug > 0)
359             cmn_err(CE_CONT, "!sv: nsc_max_devices = 0\n");
360         return (EAGAIN);
361     }

363     sv_devs = nsc_kmem_zalloc((sv_max_devices * sizeof (*sv_devs)),
364                               KM_NOSLEEP, sv_mem);

366     if (sv_devs == NULL) {
367         cmn_err(CE_WARN, "!sv: could not allocate sv_devs array");
368         return (ENOMEM);
369     }

371     for (i = 0; i < sv_max_devices; i++) {
372         mutex_init(&sv_devs[i].sv_olock, NULL, MUTEX_DRIVER, NULL);
373         rw_init(&sv_devs[i].sv_lock, NULL, RW_DRIVER, NULL);
374     }

376     if (sv_debug > 0)
377         cmn_err(CE_CONT, "!sv: sv_init_devs successful\n");

379     return (0);
380 }

383 static int
384 sv_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
385 {
386     int rc;

388     switch (cmd) {

390     case DDI_ATTACH:
391         sv_dip = dip;

```

```

393     if (ddi_create_minor_node(dip, "sv", S_IFCHR,
394         0, DDI_PSEUDO, 0) != DDI_SUCCESS)
395         goto failed;
397     mutex_enter(&sv_mutex);
399     sv_mem = nsc_register_mem("SV", NSC_MEM_LOCAL, 0);
400     if (sv_mem == NULL) {
401         mutex_exit(&sv_mutex);
402         goto failed;
403     }
405     rc = sv_init_devs();
406     if (rc != 0 && rc != EAGAIN) {
407         mutex_exit(&sv_mutex);
408         goto failed;
409     }
411     mutex_exit(&sv_mutex);
414     ddi_report_dev(dip);
416     sv_threads = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
417         DDI_PROP_DONTPASS | DDI_PROP_NOTPROM,
418         "sv_threads", sv_threads);
420     if (sv_debug > 0)
421         cmn_err(CE_CONT, "!sv: sv_threads=%d\n", sv_threads);
423     if (sv_threads > sv_threads_max)
424         sv_threads_max = sv_threads;
426     return (DDI_SUCCESS);
428 default:
429     return (DDI_FAILURE);
430 }
432 failed:
433     DTRACE_PROBE(sv_attach_failed);
434     (void) sv_detach(dip, DDI_DETACH);
435     return (DDI_FAILURE);
436 }
439 static int
440 sv_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
441 {
442     sv_dev_t *svp;
443     int i;
445     switch (cmd) {
447     case DDI_DETACH:
449         /*
450          * Check that everything is disabled.
451          */
453         mutex_enter(&sv_mutex);
455         if (sv_mod_status == SV_PREVENT_UNLOAD) {
456             mutex_exit(&sv_mutex);
457             DTRACE_PROBE(sv_detach_err_prevent);

```

```

458         return (DDI_FAILURE);
459     }
461     for (i = 0; sv_devs && i < sv_max_devices; i++) {
462         svp = &sv_devs[i];
464         if (svp->sv_state != SV_DISABLE) {
465             mutex_exit(&sv_mutex);
466             DTRACE_PROBE(sv_detach_err_busy);
467             return (DDI_FAILURE);
468         }
469     }
472     for (i = 0; sv_devs && i < sv_max_devices; i++) {
473         mutex_destroy(&sv_devs[i].sv_olock);
474         rw_destroy(&sv_devs[i].sv_lock);
475     }
477     if (sv_devs) {
478         nsc_kmem_free(sv_devs,
479             (sv_max_devices * sizeof (*sv_devs)));
480         sv_devs = NULL;
481     }
482     sv_max_devices = 0;
484     if (sv_mem) {
485         nsc_unregister_mem(sv_mem);
486         sv_mem = NULL;
487     }
489     mutex_exit(&sv_mutex);
491     /*
492     * Remove all minor nodes.
493     */
495     ddi_remove_minor_node(dip, NULL);
496     sv_dip = NULL;
498     return (DDI_SUCCESS);
500 default:
501     return (DDI_FAILURE);
502 }
503 }
505 static sv_maj_t *
506 sv_getmajor(const dev_t dev)
507 {
508     sv_maj_t **insert, *maj;
509     major_t umaj = getmajor(dev);
511     /*
512     * See if the hash table entry, or one of the hash chains
513     * is already allocated for this major number
514     */
515     if ((maj = sv_majors[SV_MAJOR_HASH(umaj)]) != 0) {
516         do {
517             if (maj->sm_major == umaj)
518                 return (maj);
519         } while ((maj = maj->sm_next) != 0);
520     }
522     /*
523     * If the sv_mutex is held, there is design flaw, as the only non-mutex

```

```

524     * held callers can be sv_enable() or sv_dev_to_sv()
525     * Return an error, instead of panicing the system
526     */
527     if (MUTEX_HELD(&sv_mutex)) {
528         cmn_err(CE_WARN, "!sv: could not allocate sv_maj_t");
529         return (NULL);
530     }
531
532     /*
533     * Determine where to allocate a new element in the hash table
534     */
535     mutex_enter(&sv_mutex);
536     insert = &(sv_majors[SV_MAJOR_HASH(umaj)]);
537     for (maj = *insert; maj; maj = maj->sm_next) {
538
539         /* Did another thread beat us to it? */
540         if (maj->sm_major == umaj)
541             return (maj);
542
543         /* Find a NULL insert point? */
544         if (maj->sm_next == NULL)
545             insert = &maj->sm_next;
546     }
547
548     /*
549     * Located the new insert point
550     */
551     *insert = nsc_kmem_zalloc(sizeof (*maj), KM_NOSLEEP, sv_mem);
552     if ((maj = *insert) != 0)
553         maj->sm_major = umaj;
554     else
555         cmn_err(CE_WARN, "!sv: could not allocate sv_maj_t");
556
557     mutex_exit(&sv_mutex);
558
559     return (maj);
560 }
561
562 /* ARGSUSED */
563
564 static int
565 sv_getinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
566 {
567     int rc = DDI_FAILURE;
568
569     switch (infocmd) {
570
571     case DDI_INFO_DEVT2DEVINFO:
572         *result = sv_dip;
573         rc = DDI_SUCCESS;
574         break;
575
576     case DDI_INFO_DEVT2INSTANCE:
577         /*
578          * We only have a single instance.
579          */
580         *result = 0;
581         rc = DDI_SUCCESS;
582         break;
583
584     default:
585         break;
586     }
587
588     return (rc);
589 }

```

```

592 /*
593  * Hashing of devices onto major device structures.
594  *
595  * Individual device structures are hashed onto one of the sm_hash[]
596  * buckets in the relevant major device structure.
597  *
598  * Hash insertion and deletion -must- be done with sv_mutex held. Hash
599  * searching does not require the mutex because of the sm_seq member.
600  * sm_seq is incremented on each insertion (-after- hash chain pointer
601  * manipulation) and each deletion (-before- hash chain pointer
602  * manipulation). When searching the hash chain, the seq number is
603  * checked before accessing each device structure, if the seq number has
604  * changed, then we restart the search from the top of the hash chain.
605  * If we restart more than SV_HASH_RETRY times, we take sv_mutex and search
606  * the hash chain (we are guaranteed that this search cannot be
607  * interrupted).
608  */
609
610 #define SV_HASH_RETRY    16
611
612 static sv_dev_t *
613 sv_dev_to_sv(const dev_t dev, sv_maj_t **majpp)
614 {
615     minor_t umin = getminor(dev);
616     sv_dev_t **hb, *next, *svp;
617     sv_maj_t *maj;
618     int seq;
619     int try;
620
621     /* Get major hash table */
622     maj = sv_getmajor(dev);
623     if (majpp)
624         *majpp = maj;
625     if (maj == NULL)
626         return (NULL);
627
628     if (maj->sm_inuse == 0) {
629         DTRACE_PROBE1(
630             sv_dev_to_sv_end,
631             dev_t, dev);
632         return (NULL);
633     }
634
635     hb = &(maj->sm_hash[SV_MINOR_HASH(umin)]);
636     try = 0;
637
638 retry:
639     if (try > SV_HASH_RETRY)
640         mutex_enter(&sv_mutex);
641
642     seq = maj->sm_seq;
643     for (svp = *hb; svp; svp = next) {
644         next = svp->sv_hash;
645
646         nsc_membar_stld(); /* preserve register load order */
647
648         if (maj->sm_seq != seq) {
649             DTRACE_PROBE1(sv_dev_to_sv_retry, dev_t, dev);
650             try++;
651             goto retry;
652         }
653
654         if (svp->sv_dev == dev)
655             break;

```

```

656     }
658     if (try > SV_HASH_RETRY)
659         mutex_exit(&sv_mutex);
661     return (svp);
662 }

665 /*
666  * Must be called with sv_mutex held.
667  */

669 static int
670 sv_get_state(const dev_t udev, sv_dev_t **svpp)
671 {
672     sv_dev_t **hb, **insert, *svp;
673     sv_maj_t *maj;
674     minor_t umin;
675     int i;

677     /* Get major hash table */
678     if ((maj = sv_getmajor(udev)) == NULL)
679         return (NULL);

681     /* Determine which minor hash table */
682     umin = getminor(udev);
683     hb = &(maj->sm_hash[SV_MINOR_HASH(umin)]);

685     /* look for clash */

687     insert = hb;

689     for (svp = *hb; svp; svp = svp->sv_hash) {
690         if (svp->sv_dev == udev)
691             break;

693         if (svp->sv_hash == NULL)
694             insert = &svp->sv_hash;
695     }

697     if (svp) {
698         DTRACE_PROBE1(
699             sv_get_state_enabled,
700             dev_t, udev);
701         return (SV_EENABLED);
702     }

704     /* look for spare sv_devs slot */

706     for (i = 0; i < sv_max_devices; i++) {
707         svp = &sv_devs[i];

709         if (svp->sv_state == SV_DISABLE)
710             break;
711     }

713     if (i >= sv_max_devices) {
714         DTRACE_PROBE1(
715             sv_get_state_noslots,
716             dev_t, udev);
717         return (SV_ENOSLOTS);
718     }

720     svp->sv_state = SV_PENDING;
721     svp->sv_pending = curthread;

```

```

723     *insert = svp;
724     svp->sv_hash = NULL;
725     maj->sm_seq++; /* must be after the store to the hash chain */

727     *svpp = svp;

729     /*
730     * We do not know the size of the underlying device at
731     * this stage, so initialise "nblocks" property to
732     * zero, and update it whenever we succeed in
733     * nsc_reserve'ing the underlying nsc_fd_t.
734     */

736     svp->sv_nblocks = 0;

738     return (0);
739 }

742 /*
743  * Remove a device structure from it's hash chain.
744  * Must be called with sv_mutex held.
745  */

747 static void
748 sv_rm_hash(sv_dev_t *svp)
749 {
750     sv_dev_t **svpp;
751     sv_maj_t *maj;

753     /* Get major hash table */
754     if ((maj = sv_getmajor(svp->sv_dev)) == NULL)
755         return;

757     /* remove svp from hash chain */

759     svpp = &(maj->sm_hash[SV_MINOR_HASH(getminor(svp->sv_dev))]);
760     while (*svpp) {
761         if (*svpp == svp) {
762             /*
763              * increment of sm_seq must be before the
764              * removal from the hash chain
765              */
766             maj->sm_seq++;
767             *svpp = svp->sv_hash;
768             break;
769         }

771         svpp = &(*svpp)->sv_hash;
772     }

774     svp->sv_hash = NULL;
775 }

777 /*
778  * Free (disable) a device structure.
779  * Must be called with sv_lock(RW_WRITER) and sv_mutex held, and will
780  * perform the exits during its processing.
781  */

783 static int
784 sv_free(sv_dev_t *svp, const int error)
785 {
786     struct cb_ops *cb_ops;
787     sv_maj_t *maj;

```

```

789  /* Get major hash table */
790  if ((maj = sv_getmajor(svp->sv_dev)) == NULL)
791      return (NULL);

793  svp->sv_state = SV_PENDING;
794  svp->sv_pending = curthread;

796  /*
797  * Close the fd's before removing from the hash or swapping
798  * back the cb_ops pointers so that the cache flushes before new
799  * io can come in.
800  */

802  if (svp->sv_fd) {
803      (void) nsc_close(svp->sv_fd);
804      svp->sv_fd = 0;
805  }

807  sv_rm_hash(svp);

809  if (error != SV_ESDOPEN &&
810      error != SV_ELYROPEN && --maj->sm_inuse == 0) {

812      if (maj->sm_dev_ops)
813          cb_ops = maj->sm_dev_ops->devo_cb_ops;
814      else
815          cb_ops = NULL;

817      if (cb_ops && maj->sm_strategy != NULL) {
818          cb_ops->cb_strategy = maj->sm_strategy;
819          cb_ops->cb_close = maj->sm_close;
820          cb_ops->cb_ioctl = maj->sm_ioctl;
821          cb_ops->cb_write = maj->sm_write;
822          cb_ops->cb_open = maj->sm_open;
823          cb_ops->cb_read = maj->sm_read;
824          cb_ops->cb_flag = maj->sm_flag;

826          if (maj->sm_awrite)
827              cb_ops->cb_awrite = maj->sm_awrite;

829          if (maj->sm_aread)
830              cb_ops->cb_aread = maj->sm_aread;

832          /*
833          * corbin XXX
834          * Leave backing device ops in maj->sm_*
835          * to handle any requests that might come
836          * in during the disable. This could be
837          * a problem however if the backing device
838          * driver is changed while we process these
839          * requests.
840          *
841          * maj->sm_strategy = 0;
842          * maj->sm_awrite = 0;
843          * maj->sm_write = 0;
844          * maj->sm_ioctl = 0;
845          * maj->sm_close = 0;
846          * maj->sm_aread = 0;
847          * maj->sm_read = 0;
848          * maj->sm_open = 0;
849          * maj->sm_flag = 0;
850          *
851          */
852      }

```

```

854          if (maj->sm_dev_ops) {
855              maj->sm_dev_ops = 0;
856          }
857      }

859  if (svp->sv_lh) {
860      cred_t *crp = ddi_get_cred();

862      /*
863      * Close the protective layered driver open using the
864      * Sun Private layered driver i/f.
865      */

867      (void) ldi_close(svp->sv_lh, FREAD|FWRITE, crp);
868      svp->sv_lh = NULL;
869  }

871  svp->sv_timestamp = nsc_lbolt();
872  svp->sv_state = SV_DISABLE;
873  svp->sv_pending = NULL;
874  rw_exit(&svp->sv_lock);
875  mutex_exit(&sv_mutex);

877  return (error);
878 }

880 /*
881 * Reserve the device, taking into account the possibility that
882 * the reserve might have to be retried.
883 */
884 static int
885 sv_reserve(nsc_fd_t *fd, int flags)
886 {
887     int eintr_count;
888     int rc;

890     eintr_count = 0;
891     do {
892         rc = nsc_reserve(fd, flags);
893         if (rc == EINTR) {
894             ++eintr_count;
895             delay(2);
896         }
897     } while ((rc == EINTR) && (eintr_count < MAX_EINTR_COUNT));

899     return (rc);
900 }

902 static int
903 sv_enable(const caddr_t path, const int flag,
904           const dev_t udev, spcs_s_info_t kstatus)
905 {
906     struct dev_ops *dev_ops;
907     struct cb_ops *cb_ops;
908     sv_dev_t *svp;
909     sv_maj_t *maj;
910     nsc_size_t nblocks;
911     int rc;
912     cred_t *crp;
913     ldi_ident_t li;

915     if (udev == (dev_t)-1 || udev == 0) {
916         DTRACE_PROBE1(
917             sv_enable_err_baddev,
918             dev_t, udev);
919         return (SV_EBADDEV);

```

```

920     }
921
922     if ((flag & ~(NSC_CACHE|NSC_DEVICE)) != 0) {
923         DTRACE_PROBE1(sv_enable_err_amode, dev_t, udev);
924         return (SV_EAMODE);
925     }
926
927     /* Get major hash table */
928     if ((maj = sv_getmajor(udev)) == NULL)
929         return (SV_EBADDEV);
930
931     mutex_enter(&sv_mutex);
932
933     rc = sv_get_state(udev, &svp);
934     if (rc) {
935         mutex_exit(&sv_mutex);
936         DTRACE_PROBE1(sv_enable_err_state, dev_t, udev);
937         return (rc);
938     }
939
940     rw_enter(&svp->sv_lock, RW_WRITER);
941
942     /*
943      * Get real fd used for io
944      */
945
946     svp->sv_dev = udev;
947     svp->sv_flag = flag;
948
949     /*
950      * OR in NSC_DEVICE to ensure that nskern grabs the real strategy
951      * function pointer before sv swaps them out.
952      */
953
954     svp->sv_fd = nsc_open(path, (svp->sv_flag | NSC_DEVICE),
955         sv_fd_def, (blind_t)udev, &rc);
956
957     if (svp->sv_fd == NULL) {
958         if (kstatus)
959             spcs_s_add(kstatus, rc);
960         DTRACE_PROBE1(sv_enable_err_fd, dev_t, udev);
961         return (sv_free(svp, SV_ESDOPEN));
962     }
963
964     /*
965      * Perform a layered driver open using the Sun Private layered
966      * driver i/f to ensure that the cb_ops structure for the driver
967      * is not detached out from under us whilst sv is enabled.
968      *
969      */
970
971     crp = ddi_get_cred();
972     svp->sv_lh = NULL;
973
974     if ((rc = ldi_ident_from_dev(svp->sv_dev, &li)) == 0) {
975         rc = ldi_open_by_dev(&svp->sv_dev,
976             OTYP_BLK, FREAD|FWRITE, crp, &svp->sv_lh, li);
977     }
978
979     if (rc != 0) {
980         if (kstatus)
981             spcs_s_add(kstatus, rc);
982         DTRACE_PROBE1(sv_enable_err_lyr_open, dev_t, udev);
983         return (sv_free(svp, SV_ELYROPEN));
984     }

```

```

986     /*
987      * Do layering if required - must happen after nsc_open().
988      */
989
990     if (maj->sm_inuse++ == 0) {
991         maj->sm_dev_ops = nsc_get_devops(getmajor(udev));
992
993         if (maj->sm_dev_ops == NULL ||
994             maj->sm_dev_ops->devo_cb_ops == NULL) {
995             DTRACE_PROBE1(sv_enable_err_load, dev_t, udev);
996             return (sv_free(svp, SV_ELOAD));
997         }
998
999         dev_ops = maj->sm_dev_ops;
1000        cb_ops = dev_ops->devo_cb_ops;
1001
1002        if (cb_ops->cb_strategy == NULL ||
1003            cb_ops->cb_strategy == nodev ||
1004            cb_ops->cb_strategy == nulldev) {
1005            DTRACE_PROBE1(sv_enable_err_nostrategy, dev_t, udev);
1006            return (sv_free(svp, SV_ELOAD));
1007        }
1008
1009        if (cb_ops->cb_strategy == sv_lyr_strategy) {
1010            DTRACE_PROBE1(sv_enable_err_svstrategy, dev_t, udev);
1011            return (sv_free(svp, SV ESTRATEGY));
1012        }
1013
1014        maj->sm_strategy = cb_ops->cb_strategy;
1015        maj->sm_close = cb_ops->cb_close;
1016        maj->sm_ioctl = cb_ops->cb_ioctl;
1017        maj->sm_write = cb_ops->cb_write;
1018        maj->sm_open = cb_ops->cb_open;
1019        maj->sm_read = cb_ops->cb_read;
1020        maj->sm_flag = cb_ops->cb_flag;
1021
1022        cb_ops->cb_flag = cb_ops->cb_flag | D_MP;
1023        cb_ops->cb_strategy = sv_lyr_strategy;
1024        cb_ops->cb_close = sv_lyr_close;
1025        cb_ops->cb_ioctl = sv_lyr_ioctl;
1026        cb_ops->cb_write = sv_lyr_write;
1027        cb_ops->cb_open = sv_lyr_open;
1028        cb_ops->cb_read = sv_lyr_read;
1029
1030        /*
1031         * Check that the driver has async I/O entry points
1032         * before changing them.
1033         */
1034
1035        if (dev_ops->devo_rev < 3 || cb_ops->cb_rev < 1) {
1036            maj->sm_awrite = 0;
1037            maj->sm_aread = 0;
1038        } else {
1039            maj->sm_awrite = cb_ops->cb_awrite;
1040            maj->sm_aread = cb_ops->cb_aread;
1041
1042            cb_ops->cb_awrite = sv_lyr_awrite;
1043            cb_ops->cb_aread = sv_lyr_aread;
1044        }
1045
1046        /*
1047         * Bug 4645743
1048         *
1049         * Prevent sv from ever unloading after it has interposed
1050         * on a major device because there is a race between
1051         * sv removing its layered entry points from the target

```

```

1052     * dev_ops, a client coming in and accessing the driver,
1053     * and the kernel modunloading the sv text.
1054     *
1055     * To allow unload, do svboot -u, which only happens in
1056     * pkgmgr time.
1057     */
1058     ASSERT(MUTEX_HELD(&sv_mutex));
1059     sv_mod_status = SV_PREVENT_UNLOAD;
1060 }

1063     svp->sv_timestamp = nsc_lbolt();
1064     svp->sv_state = SV_ENABLE;
1065     svp->sv_pending = NULL;
1066     rw_exit(&svp->sv_lock);

1068     sv_ndevices++;
1069     mutex_exit(&sv_mutex);

1071     nblocks = 0;
1072     if (sv_reserve(svp->sv_fd, NSC_READ|NSC_MULTI|NSC_PCATCH) == 0) {
1073         nblocks = svp->sv_nblocks;
1074         nsc_release(svp->sv_fd);
1075     }

1077     cmn_err(CE_CONT, "!sv: rdev 0x%lx, nblocks %" NSC_SZFMT "\n",
1078             svp->sv_dev, nblocks);

1080     return (0);
1081 }

1084 static int
1085 sv_prepare_unload()
1086 {
1087     int rc = 0;

1089     mutex_enter(&sv_mutex);

1091     if (sv_mod_status == SV_PREVENT_UNLOAD) {
1092         if ((sv_ndevices != 0) || (sv_tset != NULL)) {
1093             rc = EBUSY;
1094         } else {
1095             sv_mod_status = SV_ALLOW_UNLOAD;
1096             delay(SV_WAIT_UNLOAD * drv_usecstohz(1000000));
1097         }
1098     }

1100     mutex_exit(&sv_mutex);
1101     return (rc);
1102 }

1104 static int
1105 svattach_fd(blind_t arg)
1106 {
1107     dev_t dev = (dev_t)arg;
1108     sv_dev_t *svp = sv_dev_to_sv(dev, NULL);
1109     int rc;

1111     if (sv_debug > 0)
1112         cmn_err(CE_CONT, "!svattach_fd(%p, %p)\n", arg, (void *)svp);

1114     if (svp == NULL) {
1115         cmn_err(CE_WARN, "!svattach_fd: no state (arg %p)", arg);
1116         return (0);
1117     }

```

```

1119     if ((rc = nsc_partsize(svp->sv_fd, &svp->sv_nblocks)) != 0) {
1120         cmn_err(CE_WARN,
1121             "!svattach_fd: nsc_partsize() failed, rc %d", rc);
1122         svp->sv_nblocks = 0;
1123     }

1125     if ((rc = nsc_maxfbas(svp->sv_fd, 0, &svp->sv_maxfbas)) != 0) {
1126         cmn_err(CE_WARN,
1127             "!svattach_fd: nsc_maxfbas() failed, rc %d", rc);
1128         svp->sv_maxfbas = 0;
1129     }

1131     if (sv_debug > 0) {
1132         cmn_err(CE_CONT,
1133             "!svattach_fd(%p): size %" NSC_SZFMT ", "
1134             "maxfbas %" NSC_SZFMT "\n",
1135             arg, svp->sv_nblocks, svp->sv_maxfbas);
1136     }

1138     return (0);
1139 }

1142 static int
1143 svdetach_fd(blind_t arg)
1144 {
1145     dev_t dev = (dev_t)arg;
1146     sv_dev_t *svp = sv_dev_to_sv(dev, NULL);

1148     if (sv_debug > 0)
1149         cmn_err(CE_CONT, "!svdetach_fd(%p, %p)\n", arg, (void *)svp);

1151     /* svp can be NULL during disable of an sv */
1152     if (svp == NULL)
1153         return (0);

1155     svp->sv_maxfbas = 0;
1156     svp->sv_nblocks = 0;
1157     return (0);
1158 }

1161 /*
1162  * Side effect: if called with (guard != 0), then expects both sv_mutex
1163  * and sv_lock(RW_WRITER) to be held, and will release them before returning.
1164  */

1166 /* ARGSUSED */
1167 static int
1168 sv_disable(dev_t dev, spcs_s_info_t kstatus)
1169 {
1170     sv_dev_t *svp = sv_dev_to_sv(dev, NULL);

1172     if (svp == NULL) {

1174         DTRACE_PROBE1(sv_disable_err_nodev, sv_dev_t *, svp);
1175         return (SV_ENODEV);
1176     }

1178     mutex_enter(&sv_mutex);
1179     rw_enter(&svp->sv_lock, RW_WRITER);

1181     if (svp->sv_fd == NULL || svp->sv_state != SV_ENABLE) {
1182         rw_exit(&svp->sv_lock);
1183         mutex_exit(&sv_mutex);

```



```

1185         DTRACE_PROBE1(sv_disable_err_disabled, sv_dev_t *, svp);
1186         return (SV_EDISABLED);
1187     }

1190     sv_ndevices--;
1191     return (sv_free(svp, 0));
1192 }

1196 static int
1197 sv_lyr_open(dev_t *devp, int flag, int otyp, cred_t *crp)
1198 {
1199     nsc_buf_t *tmp;
1200     sv_dev_t *svp;
1201     sv_maj_t *maj;
1202     int (*fn)();
1203     dev_t odev;
1204     int ret;
1205     int rc;

1207     svp = sv_dev_to_sv(*devp, &maj);

1209     if (svp) {
1210         if (svp->sv_state == SV_PENDING &&
1211             svp->sv_pending == curthread) {
1212             /*
1213              * This is a recursive open from a call to
1214              * ddi_lyr_open_by_devt and so we just want
1215              * to pass it straight through to the
1216              * underlying driver.
1217              */
1218             DTRACE_PROBE2(sv_lyr_open_recursive,
1219                 sv_dev_t *, svp,
1220                 dev_t, *devp);
1221             svp = NULL;
1222         } else
1223             rw_enter(&svp->sv_lock, RW_READER);
1224     }

1226     odev = *devp;

1228     if (maj && (fn = maj->sm_open) != 0) {
1229         if (!(maj->sm_flag & D_MP)) {
1230             UNSAFE_ENTER();
1231             ret = (*fn)(devp, flag, otyp, crp);
1232             UNSAFE_EXIT();
1233         } else {
1234             ret = (*fn)(devp, flag, otyp, crp);
1235         }

1237         if (ret == 0) {
1238             /*
1239              * Re-acquire svp if the driver changed *devp.
1240              */

1242             if (*devp != odev) {
1243                 if (svp != NULL)
1244                     rw_exit(&svp->sv_lock);

1246                 svp = sv_dev_to_sv(*devp, NULL);

1248                 if (svp) {
1249                     rw_enter(&svp->sv_lock, RW_READER);

```

```

1250         }
1251     }
1252     } else {
1253         ret = ENODEV;
1254     }
1255 }

1257     if (svp && ret != 0 && svp->sv_state == SV_ENABLE) {
1258         /*
1259          * Underlying DDI open failed, but we have this
1260          * device SV enabled. If we can read some data
1261          * from the device, fake a successful open (this
1262          * probably means that this device is RDC'd and we
1263          * are getting the data from the secondary node).
1264          *
1265          * The reserve must be done with NSC_TRY|NSC_NOWAIT to
1266          * ensure that it does not deadlock if this open is
1267          * coming from nskernel:get_bsize().
1268          */
1269         rc = sv_reserve(svp->sv_fd,
1270             NSC_TRY | NSC_NOWAIT | NSC_MULT | NSC_PCATCH);
1271         if (rc == 0) {
1272             tmp = NULL;

1274             rc = nsc_alloc_buf(svp->sv_fd, 0, 1, NSC_READ, &tmp);
1275             if (rc <= 0) {
1276                 /* success */
1277                 ret = 0;
1278             }

1280             if (tmp) {
1281                 (void) nsc_free_buf(tmp);
1282                 tmp = NULL;
1283             }

1285             nsc_release(svp->sv_fd);

1287             /*
1288              * Count the number of layered opens that we
1289              * fake since we have to fake a matching number
1290              * of closes (OTYP_LYR open/close calls must be
1291              * paired).
1292              */

1294             if (ret == 0 && otyp == OTYP_LYR) {
1295                 mutex_enter(&svp->sv_olock);
1296                 svp->sv_openlcnt++;
1297                 mutex_exit(&svp->sv_olock);
1298             }
1299         }
1300     }

1302     if (svp) {
1303         rw_exit(&svp->sv_lock);
1304     }

1306     return (ret);
1307 }

1310 static int
1311 sv_lyr_close(dev_t dev, int flag, int otyp, cred_t *crp)
1312 {
1313     sv_dev_t *svp;
1314     sv_maj_t *maj;
1315     int (*fn)();

```

```

1316     int ret;
1318     svp = sv_dev_to_sv(dev, &maj);
1320     if (svp &&
1321         svp->sv_state == SV_PENDING &&
1322         svp->sv_pending == curthread) {
1323         /*
1324          * This is a recursive open from a call to
1325          * ddi_lyr_close and so we just want
1326          * to pass it straight through to the
1327          * underlying driver.
1328          */
1329         DTRACE_PROBE2(sv_lyr_close_recursive, sv_dev_t *, svp,
1330                     dev_t, dev);
1331         svp = NULL;
1332     }
1334     if (svp) {
1335         rw_enter(&svp->sv_lock, RW_READER);
1337         if (otyp == OTYP_LYR) {
1338             mutex_enter(&svp->sv_olock);
1340             if (svp->sv_openlcnt) {
1341                 /*
1342                  * Consume sufficient layered closes to
1343                  * account for the opens that we faked
1344                  * whilst the device was failed.
1345                  */
1346                 svp->sv_openlcnt--;
1347                 mutex_exit(&svp->sv_olock);
1348                 rw_exit(&svp->sv_lock);
1350                 DTRACE_PROBE1(sv_lyr_close_end, dev_t, dev);
1352                 return (0);
1353             }
1355             mutex_exit(&svp->sv_olock);
1356         }
1357     }
1359     if (maj && (fn = maj->sm_close) != 0) {
1360         if (!(maj->sm_flag & D_MP)) {
1361             UNSAFE_ENTER();
1362             ret = (*fn)(dev, flag, otyp, crp);
1363             UNSAFE_EXIT();
1364         } else {
1365             ret = (*fn)(dev, flag, otyp, crp);
1366         }
1367     } else {
1368         ret = ENODEV;
1369     }
1371     if (svp) {
1372         rw_exit(&svp->sv_lock);
1373     }
1375     return (ret);
1376 }
1379 /*
1380 * Convert the specified dev_t into a locked and enabled sv_dev_t, or
1381 * return NULL.

```

```

1382 */
1383 static sv_dev_t *
1384 sv_find_enabled(const dev_t dev, sv_maj_t **majpp)
1385 {
1386     sv_dev_t *svp;
1388     while ((svp = sv_dev_to_sv(dev, majpp)) != NULL) {
1389         rw_enter(&svp->sv_lock, RW_READER);
1391         if (svp->sv_state == SV_ENABLE) {
1392             /* locked and enabled */
1393             break;
1394         }
1396         /*
1397          * State was changed while waiting on the lock.
1398          * Wait for a stable state.
1399          */
1400         rw_exit(&svp->sv_lock);
1402         DTRACE_PROBE1(sv_find_enabled_retry, dev_t, dev);
1404         delay(2);
1405     }
1407     return (svp);
1408 }
1411 static int
1412 sv_lyr_uio(dev_t dev, uio_t *uiop, cred_t *crp, int rw)
1413 {
1414     sv_dev_t *svp;
1415     sv_maj_t *maj;
1416     int (*fn)();
1417     int rc;
1419     svp = sv_find_enabled(dev, &maj);
1420     if (svp == NULL) {
1421         if (maj) {
1422             if (rw == NSC_READ)
1423                 fn = maj->sm_read;
1424             else
1425                 fn = maj->sm_write;
1427             if (fn != 0) {
1428                 if (!(maj->sm_flag & D_MP)) {
1429                     UNSAFE_ENTER();
1430                     rc = (*fn)(dev, uiop, crp);
1431                     UNSAFE_EXIT();
1432                 } else {
1433                     rc = (*fn)(dev, uiop, crp);
1434                 }
1435             }
1437             return (rc);
1438         } else {
1439             return (ENODEV);
1440         }
1441     }
1443     ASSERT(RW_READ_HELD(&svp->sv_lock));
1445     if (svp->sv_flag == 0) {
1446         /*
1447          * guard access mode

```

```

1448     * - prevent user level access to the device
1449     */
1450     DTRACE_PROBE1(sv_lyr_uio_err_guard, uio_t *, uiop);
1451     rc = EPERM;
1452     goto out;
1453 }
1454
1455 if ((rc = sv_reserve(svp->sv_fd, NSC_MULTI|NSC_PCATCH)) != 0) {
1456     DTRACE_PROBE1(sv_lyr_uio_err_rsrv, uio_t *, uiop);
1457     goto out;
1458 }
1459
1460 if (rw == NSC_READ)
1461     rc = nsc_uread(svp->sv_fd, uiop, crp);
1462 else
1463     rc = nsc_uwrite(svp->sv_fd, uiop, crp);
1464
1465 nsc_release(svp->sv_fd);
1466
1467 out:
1468     rw_exit(&svp->sv_lock);
1469
1470     return (rc);
1471 }
1472
1473
1474 static int
1475 sv_lyr_read(dev_t dev, uio_t *uiop, cred_t *crp)
1476 {
1477     return (sv_lyr_uio(dev, uiop, crp, NSC_READ));
1478 }
1479
1480
1481 static int
1482 sv_lyr_write(dev_t dev, uio_t *uiop, cred_t *crp)
1483 {
1484     return (sv_lyr_uio(dev, uiop, crp, NSC_WRITE));
1485 }
1486
1487
1488 /* ARGSUSED */
1489
1490 static int
1491 sv_lyr_aread(dev_t dev, struct aio_req *aio, cred_t *crp)
1492 {
1493     return (aphysio(sv_lyr_strategy,
1494         anocancel, dev, B_READ, minphys, aio));
1495 }
1496
1497
1498 /* ARGSUSED */
1499
1500 static int
1501 sv_lyr_awrite(dev_t dev, struct aio_req *aio, cred_t *crp)
1502 {
1503     return (aphysio(sv_lyr_strategy,
1504         anocancel, dev, B_WRITE, minphys, aio));
1505 }
1506
1507
1508 /*
1509  * Set up an array containing the list of raw path names
1510  * The array for the paths is svl and the size of the array is
1511  * in size.
1512  *
1513  * If there are more layered devices than will fit in the array,

```

```

1514  * the number of extra layered devices is returned. Otherwise
1515  * zero is return.
1516  */
1517  * Input:
1518  *     svn      : array for paths
1519  *     size     : size of the array
1520  *
1521  * Output (extra):
1522  *     zero     : All paths fit in array
1523  *     >0      : Number of defined layered devices don't fit in array
1524  */
1525
1526 static int
1527 sv_list(void *ptr, const int size, int *extra, const int ilp32)
1528 {
1529     sv_name32_t *svn32;
1530     sv_name_t *svn;
1531     sv_dev_t *svp;
1532     int *mode, *nblocks;
1533     int i, index;
1534     char *path;
1535
1536     *extra = 0;
1537     index = 0;
1538
1539     if (ilp32)
1540         svn32 = ptr;
1541     else
1542         svn = ptr;
1543
1544     mutex_enter(&sv_mutex);
1545     for (i = 0; i < sv_max_devices; i++) {
1546         svp = &sv_devs[i];
1547
1548         rw_enter(&svp->sv_lock, RW_READER);
1549
1550         if (svp->sv_state != SV_ENABLE) {
1551             rw_exit(&svp->sv_lock);
1552             continue;
1553         }
1554
1555         if ((*extra) != 0 || ptr == NULL) {
1556             /* Another overflow entry */
1557             rw_exit(&svp->sv_lock);
1558             (*extra)++;
1559             continue;
1560         }
1561
1562         if (ilp32) {
1563             nblocks = &svn32->svn_nblocks;
1564             mode = &svn32->svn_mode;
1565             path = svn32->svn_path;
1566
1567             svn32->svn_timestamp = (uint32_t)svp->sv_timestamp;
1568             svn32++;
1569         } else {
1570             nblocks = &svn->svn_nblocks;
1571             mode = &svn->svn_mode;
1572             path = svn->svn_path;
1573
1574             svn->svn_timestamp = svp->sv_timestamp;
1575             svn++;
1576         }
1577
1578         (void) strcpy(path, nsc_pathname(svp->sv_fd));
1579         *nblocks = svp->sv_nblocks;

```

```

1580     *mode = svp->sv_flag;
1582     if (*nblocks == 0) {
1583         if (sv_debug > 3)
1584             cmn_err(CE_CONT, "!sv_list: need to reserve\n");
1586         if (sv_reserve(svp->sv_fd, NSC_MULTI|NSC_PCATCH) == 0) {
1587             *nblocks = svp->sv_nblocks;
1588             nsc_release(svp->sv_fd);
1589         }
1590     }
1592     if (++index >= size) {
1593         /* Out of space */
1594         (*extra)++;
1595     }
1597     rw_exit(&svp->sv_lock);
1598 }
1599 mutex_exit(&sv_mutex);
1601 if (index < size) {
1602     /* NULL terminated list */
1603     if (ilp32)
1604         svn32->svn_path[0] = '\0';
1605     else
1606         svn->svn_path[0] = '\0';
1607 }
1609 return (0);
1610 }
1613 static void
1614 sv_thread_tune(int threads)
1615 {
1616     int incr = (threads > 0) ? 1 : -1;
1617     int change = 0;
1618     int nthreads;
1620     ASSERT(MUTEX_HELD(&sv_mutex));
1622     if (sv_threads_extra) {
1623         /* keep track of any additional threads requested */
1624         if (threads > 0) {
1625             sv_threads_extra += threads;
1626             return;
1627         }
1628         threads = -threads;
1629         if (threads >= sv_threads_extra) {
1630             threads -= sv_threads_extra;
1631             sv_threads_extra = 0;
1632             /* fall through to while loop */
1633         } else {
1634             sv_threads_extra -= threads;
1635             return;
1636         }
1637     } else if (threads > 0) {
1638         /*
1639          * do not increase the number of threads beyond
1640          * sv_threads_max when doing dynamic thread tuning
1641          */
1642         nthreads = nst_nthread(sv_tset);
1643         if ((nthreads + threads) > sv_threads_max) {
1644             sv_threads_extra = nthreads + threads - sv_threads_max;
1645             threads = sv_threads_max - nthreads;

```

```

1646         if (threads <= 0)
1647             return;
1648     }
1649 }
1651 if (threads < 0)
1652     threads = -threads;
1654 while (threads-- > 0) {
1655     nthreads = nst_nthread(sv_tset);
1656     sv_threads_needed += incr;
1658     if (sv_threads_needed >= nthreads)
1659         change += nst_add_thread(sv_tset, sv_threads_inc);
1660     else if ((sv_threads_needed <
1661             (nthreads - (sv_threads_inc + sv_threads_hysteresis))) &&
1662             ((nthreads - sv_threads_inc) >= sv_threads))
1663         change -= nst_del_thread(sv_tset, sv_threads_inc);
1664 }
1666 #ifndef DEBUG
1667     if (change) {
1668         cmn_err(CE_NOTE,
1669             "!sv_thread_tune: threads needed %d, nthreads %d, "
1670             "nthreads change %d",
1671             sv_threads_needed, nst_nthread(sv_tset), change);
1672     }
1673 #endif
1674 }
1677 /* ARGSUSED */
1678 static int
1679 svopen(dev_t *devp, int flag, int otyp, cred_t *crp)
1680 {
1681     int rc;
1683     mutex_enter(&sv_mutex);
1684     rc = sv_init_devs();
1685     mutex_exit(&sv_mutex);
1687     return (rc);
1688 }
1691 /* ARGSUSED */
1692 static int
1693 svclose(dev_t dev, int flag, int otyp, cred_t *crp)
1694 {
1695     const int secs = HZ * 5;
1696     const int ticks = HZ / 10;
1697     int loops = secs / ticks;
1699     mutex_enter(&sv_mutex);
1700     while (sv_ndevice <= 0 && sv_tset != NULL && loops > 0) {
1701         if (nst_nlive(sv_tset) <= 0) {
1702             nst_destroy(sv_tset);
1703             sv_tset = NULL;
1704             break;
1705         }
1707         /* threads still active - wait for them to exit */
1708         mutex_exit(&sv_mutex);
1709         delay(ticks);
1710         loops--;
1711         mutex_enter(&sv_mutex);

```

```

1712     }
1713     mutex_exit(&sv_mutex);

1715     if (loops <= 0) {
1716         cmn_err(CE_WARN,
1717 #ifndef DEBUG
1718             /* do not write to console when non-DEBUG */
1719             "!")
1720 #endif
1721         "sv:svclose: threads still active "
1722         "after %d sec - leaking thread set", secs);
1723     }

1725     return (0);
1726 }

1729 static int
1730 svioctl(dev_t dev, int cmd, intp_t arg, int mode, cred_t *crp, int *rvalp)
1731 {
1732     char itmp1[12], itmp2[12]; /* temp char array for editing ints */
1733     spcs_s_info_t kstatus; /* Kernel version of spcs status */
1734     spcs_s_info_t ustatus; /* Address of user version of spcs status */
1735     sv_list32_t svl32; /* 32 bit Initial structure for SVIOC_LIST */
1736     sv_version_t svv; /* Version structure */
1737     sv_conf_t svc; /* User config structure */
1738     sv_list_t svl; /* Initial structure for SVIOC_LIST */
1739     void *usvn; /* Address of user sv_name_t */
1740     void *svn = NULL; /* Array for SVIOC_LIST */
1741     uint64_t phash; /* pathname hash */
1742     int rc = 0; /* Return code -- errno */
1743     int size; /* Number of items in array */
1744     int bytes; /* Byte size of array */
1745     int ilp32; /* Convert data structures for ilp32 userland */

1747     *rvalp = 0;

1749     /*
1750     * If sv_mod_status is 0 or SV_PREVENT_UNLOAD, then it will continue.
1751     * else it means it previously was SV_PREVENT_UNLOAD, and now it's
1752     * SV_ALLOW_UNLOAD, expecting the driver to eventually unload.
1753     *
1754     * SV_ALLOW_UNLOAD is final state, so no need to grab sv_mutex.
1755     */
1756     if (sv_mod_status == SV_ALLOW_UNLOAD) {
1757         return (EBUSY);
1758     }

1760     if ((cmd != SVIOC_LIST) && ((rc = drv_priv(crp)) != 0))
1761         return (rc);

1763     kstatus = spcs_s_kcreate();
1764     if (!kstatus) {
1765         DTRACE_PROBE1(sv_ioctl_err_kcreate, dev_t, dev);
1766         return (ENOMEM);
1767     }

1769     ilp32 = (ddi_model_convert_from((mode & FMODELS)) == DDI_MODEL_ILP32);

1771     switch (cmd) {

1773     case SVIOC_ENABLE:

1775         if (ilp32) {
1776             sv_conf32_t svc32;

```

```

1778         if (ddi_copyin((void *)arg, &svc32,
1779             sizeof (svc32), mode) < 0) {
1780             spcs_s_kfree(kstatus);
1781             return (EFAULT);
1782         }

1784         svc.svc_error = (spcs_s_info_t)svc32.svc_error;
1785         (void) strcpy(svc.svc_path, svc32.svc_path);
1786         svc.svc_flag = svc32.svc_flag;
1787         svc.svc_major = svc32.svc_major;
1788         svc.svc_minor = svc32.svc_minor;
1789     } else {
1790         if (ddi_copyin((void *)arg, &svc,
1791             sizeof (svc), mode) < 0) {
1792             spcs_s_kfree(kstatus);
1793             return (EFAULT);
1794         }
1795     }

1797     /* force to raw access */
1798     svc.svc_flag = NSC_DEVICE;

1800     if (sv_tset == NULL) {
1801         mutex_enter(&sv_mutex);

1803         if (sv_tset == NULL) {
1804             sv_tset = nst_init("sv_thr", sv_threads);
1805         }

1807         mutex_exit(&sv_mutex);

1809         if (sv_tset == NULL) {
1810             cmn_err(CE_WARN,
1811                 "!sv: could not allocate %d threads",
1812                 sv_threads);
1813         }
1814     }

1816     rc = sv_enable(svc.svc_path, svc.svc_flag,
1817         makedevice(svc.svc_major, svc.svc_minor), kstatus);

1819     if (rc == 0) {
1820         sv_config_time = nsc_lbolt();

1822         mutex_enter(&sv_mutex);
1823         sv_thread_tune(sv_threads_dev);
1824         mutex_exit(&sv_mutex);
1825     }

1827     DTRACE_PROBE3(sv_ioctl_end, dev_t, dev, int, *rvalp, int, rc);

1829     return (spcs_s_ocopyoutf(&kstatus, svc.svc_error, rc));
1830     /* NOTREACHED */

1832     case SVIOC_DISABLE:

1834         if (ilp32) {
1835             sv_conf32_t svc32;

1837             if (ddi_copyin((void *)arg, &svc32,
1838                 sizeof (svc32), mode) < 0) {
1839                 spcs_s_kfree(kstatus);
1840                 return (EFAULT);
1841             }

1843             svc.svc_error = (spcs_s_info_t)svc32.svc_error;

```

```

1844     svc.svc_major = svc32.svc_major;
1845     svc.svc_minor = svc32.svc_minor;
1846     (void) strcpy(svc.svc_path, svc32.svc_path);
1847     svc.svc_flag = svc32.svc_flag;
1848 } else {
1849     if (ddi_copyin((void *)arg, &svc,
1850                 sizeof (svc), mode) < 0) {
1851         spcs_s_kfree(kstatus);
1852         return (EFAULT);
1853     }
1854 }
1855
1856 if (svc.svc_major == (major_t)-1 &&
1857     svc.svc_minor == (minor_t)-1) {
1858     sv_dev_t *svp;
1859     int i;
1860
1861     /*
1862      * User level could not find the minor device
1863      * node, so do this the slow way by searching
1864      * the entire sv config for a matching pathname.
1865      */
1866
1867     phash = nsc_strhash(svc.svc_path);
1868
1869     mutex_enter(&sv_mutex);
1870
1871     for (i = 0; i < sv_max_devices; i++) {
1872         svp = &sv_devs[i];
1873
1874         if (svp->sv_state == SV_DISABLE ||
1875             svp->sv_fd == NULL)
1876             continue;
1877
1878         if (nsc_fdpathcmp(svp->sv_fd, phash,
1879                         svc.svc_path) == 0) {
1880             svc.svc_major = getmajor(svp->sv_dev);
1881             svc.svc_minor = getminor(svp->sv_dev);
1882             break;
1883         }
1884     }
1885
1886     mutex_exit(&sv_mutex);
1887
1888     if (svc.svc_major == (major_t)-1 &&
1889         svc.svc_minor == (minor_t)-1)
1890         return (spcs_s_ocopyoutf(&kstatus,
1891                                 svc.svc_error, SV_ENODEV));
1892 }
1893
1894 rc = sv_disable(makedevice(svc.svc_major, svc.svc_minor),
1895               kstatus);
1896
1897 if (rc == 0) {
1898     sv_config_time = nsc_lbolt();
1899
1900     mutex_enter(&sv_mutex);
1901     sv_thread_tune(-sv_threads_dev);
1902     mutex_exit(&sv_mutex);
1903 }
1904
1905 DTRACE_PROBE3(sv_ioctl_2, dev_t, dev, int, *rvalp, int, rc);
1906
1907 return (spcs_s_ocopyoutf(&kstatus, svc.svc_error, rc));
1908 /* NOTREACHED */

```

```

1910     case SVIOC_LIST:
1911
1912         if (ilp32) {
1913             if (ddi_copyin((void *)arg, &svl32,
1914                         sizeof (svl32), mode) < 0) {
1915                 spcs_s_kfree(kstatus);
1916                 return (EFAULT);
1917             }
1918
1919             uestatus = (spcs_s_info_t)svl32.svl_error;
1920             size = svl32.svl_count;
1921             usvn = (void *) (unsigned long)svl32.svl_names;
1922         } else {
1923             if (ddi_copyin((void *)arg, &svl,
1924                         sizeof (svl), mode) < 0) {
1925                 spcs_s_kfree(kstatus);
1926                 return (EFAULT);
1927             }
1928
1929             uestatus = svl.svl_error;
1930             size = svl.svl_count;
1931             usvn = svl.svl_names;
1932         }
1933
1934         /* Do some boundary checking */
1935         if ((size < 0) || (size > sv_max_devices)) {
1936             /* Array size is out of range */
1937             return (spcs_s_ocopyoutf(&kstatus, uestatus,
1938                                     SV_EARBOUNDS, "0",
1939                                     spcs_s_inttostring(sv_max_devices, itmpl,
1940                                                         sizeof (itmpl), 0),
1941                                     spcs_s_inttostring(size, itmp2,
1942                                                         sizeof (itmp2), 0)));
1943         }
1944
1945         if (ilp32)
1946             bytes = size * sizeof (sv_name32_t);
1947         else
1948             bytes = size * sizeof (sv_name_t);
1949
1950         /* Allocate memory for the array of structures */
1951         if (bytes != 0) {
1952             svn = kmem_zalloc(bytes, KM_SLEEP);
1953             if (!svn) {
1954                 return (spcs_s_ocopyoutf(&kstatus,
1955                                         uestatus, ENOMEM));
1956             }
1957         }
1958
1959         rc = sv_list(svn, size, rvalp, ilp32);
1960         if (rc) {
1961             if (svn != NULL)
1962                 kmem_free(svn, bytes);
1963             return (spcs_s_ocopyoutf(&kstatus, uestatus, rc));
1964         }
1965
1966         if (ilp32) {
1967             svl32.svl_timestamp = (uint32_t)sv_config_time;
1968             svl32.svl_maxdevs = (int32_t)sv_max_devices;
1969
1970             /* Return the list structure */
1971             if (ddi_copyout(&svl32, (void *)arg,
1972                         sizeof (svl32), mode) < 0) {
1973                 spcs_s_kfree(kstatus);
1974                 if (svn != NULL)
1975                     kmem_free(svn, bytes);

```

```

1976         return (EFAULT);
1977     }
1978     } else {
1979         svl.svl_timestamp = sv_config_time;
1980         svl.svl_maxdevs = sv_max_devices;

1982         /* Return the list structure */
1983         if (ddi_copyout(&svl, (void *)arg,
1984             sizeof (svl), mode) < 0) {
1985             spcs_s_kfree(kstatus);
1986             if (svn != NULL)
1987                 kmem_free(svn, bytes);
1988             return (EFAULT);
1989         }
1990     }

1992     /* Return the array */
1993     if (svn != NULL) {
1994         if (ddi_copyout(svn, usvn, bytes, mode) < 0) {
1995             kmem_free(svn, bytes);
1996             spcs_s_kfree(kstatus);
1997             return (EFAULT);
1998         }
1999         kmem_free(svn, bytes);
2000     }

2002     DTRACE_PROBE3(sv_ioctl_3, dev_t, dev, int, *rvalp, int, 0);

2004     return (spcs_s_ocopyoutf(&kstatus, ustatus, 0));
2005     /* NOTREACHED */

2007     case SVIOC_VERSION:

2009         if (ilp32) {
2010             sv_version32_t svv32;

2012             if (ddi_copyin((void *)arg, &svv32,
2013                 sizeof (svv32), mode) < 0) {
2014                 spcs_s_kfree(kstatus);
2015                 return (EFAULT);
2016             }

2018             svv32.svv_major_rev = sv_major_rev;
2019             svv32.svv_minor_rev = sv_minor_rev;
2020             svv32.svv_micro_rev = sv_micro_rev;
2021             svv32.svv_baseline_rev = sv_baseline_rev;

2023             if (ddi_copyout(&svv32, (void *)arg,
2024                 sizeof (svv32), mode) < 0) {
2025                 spcs_s_kfree(kstatus);
2026                 return (EFAULT);
2027             }

2029             ustatus = (spcs_s_info_t)svv32.svv_error;
2030         } else {
2031             if (ddi_copyin((void *)arg, &svv,
2032                 sizeof (svv), mode) < 0) {
2033                 spcs_s_kfree(kstatus);
2034                 return (EFAULT);
2035             }

2037             svv.svv_major_rev = sv_major_rev;
2038             svv.svv_minor_rev = sv_minor_rev;
2039             svv.svv_micro_rev = sv_micro_rev;
2040             svv.svv_baseline_rev = sv_baseline_rev;

```

```

2042         if (ddi_copyout(&svv, (void *)arg,
2043             sizeof (svv), mode) < 0) {
2044             spcs_s_kfree(kstatus);
2045             return (EFAULT);
2046         }

2048         ustatus = svv.svv_error;
2049     }

2051     DTRACE_PROBE3(sv_ioctl_4, dev_t, dev, int, *rvalp, int, 0);

2053     return (spcs_s_ocopyoutf(&kstatus, ustatus, 0));
2054     /* NOTREACHED */

2056     case SVIIOC_UNLOAD:
2057         rc = sv_prepare_unload();

2059         if (ddi_copyout(&rc, (void *)arg, sizeof (rc), mode) < 0) {
2060             rc = EFAULT;
2061         }

2063         spcs_s_kfree(kstatus);
2064         return (rc);

2066     default:
2067         spcs_s_kfree(kstatus);

2069         DTRACE_PROBE3(sv_ioctl_4, dev_t, dev, int, *rvalp, int, EINVAL);

2071         return (EINVAL);
2072         /* NOTREACHED */
2073     }

2075     /* NOTREACHED */
2076 }

2079 /* ARGSUSED */
2080 static int
2081 svprint(dev_t dev, char *str)
2082 {
2083     int instance = ddi_get_instance(sv_dip);
2084     cmn_err(CE_WARN, "!%s%d: %s", ddi_get_name(sv_dip), instance, str);
2085     return (0);
2086 }

2089 static void
2090 _sv_lyr_strategy(struct buf *bp)
2091 {
2092     caddr_t buf_addr; /* pointer to linear buffer in bp */
2093     nsc_buf_t *bufh = NULL;
2094     nsc_buf_t *hndl = NULL;
2095     sv_dev_t *svp;
2096     nsc_vec_t *v;
2097     sv_maj_t *maj;
2098     nsc_size_t fba_req, fba_len; /* FBA lengths */
2099     nsc_off_t fba_off; /* FBA offset */
2100     size_t tocopy, nbytes; /* byte lengths */
2101     int rw, rc; /* flags and return codes */
2102     int (*fn)();

2104     rc = 0;

2106     if (sv_debug > 5)
2107         cmn_err(CE_CONT, "!_sv_lyr_strategy(%p)\n", (void *)bp);

```

```

2109     svp = sv_find_enabled(bp->b_edev, &maj);
2110     if (svp == NULL) {
2111         if (maj == (fn = maj->sm_strategy) != 0) {
2112             if (!(maj->sm_flag & D_MP)) {
2113                 UNSAFE_ENTER();
2114                 rc = (*fn)(bp);
2115                 UNSAFE_EXIT();
2116             } else {
2117                 rc = (*fn)(bp);
2118             }
2119             return;
2120         } else {
2121             bioerror(bp, ENODEV);
2122             biodone(bp);
2123             return;
2124         }
2125     }
2127     ASSERT(RW_READ_HELD(&svp->sv_lock));
2129     if (svp->sv_flag == 0) {
2130         /*
2131          * guard access mode
2132          * - prevent user level access to the device
2133          */
2134         DTRACE_PROBE1(sv_lyr_strategy_err_guard, struct buf *, bp);
2135         bioerror(bp, EPERM);
2136         goto out;
2137     }
2139     if ((rc = sv_reserve(svp->sv_fd, NSC_MULTI|NSC_PCATCH)) != 0) {
2140         DTRACE_PROBE1(sv_lyr_strategy_err_rsrv, struct buf *, bp);
2142         if (rc == EINTR)
2143             cmn_err(CE_WARN, "Insc_reserve() returned EINTR");
2144         bioerror(bp, rc);
2145         goto out;
2146     }
2148     if (bp->b_lblkno >= (diskaddr_t)svp->sv_nblocks) {
2149         DTRACE_PROBE1(sv_lyr_strategy_eof, struct buf *, bp);
2151         if (bp->b_flags & B_READ) {
2152             /* return EOF, not an error */
2153             bp->b_resid = bp->b_bcount;
2154             bioerror(bp, 0);
2155         } else
2156             bioerror(bp, EINVAL);
2158         goto done;
2159     }
2161     /*
2162     * Preallocate a handle once per call to strategy.
2163     * If this fails, then the nsc_alloc_buf() will allocate
2164     * a temporary handle per allocation/free pair.
2165     */
2167     DTRACE_PROBE1(sv_dbg_alloch_start, sv_dev_t *, svp);
2169     bufh = nsc_alloc_handle(svp->sv_fd, NULL, NULL, NULL);
2171     DTRACE_PROBE1(sv_dbg_alloch_end, sv_dev_t *, svp);
2173     if (bufh && (bufh->sb_flag & NSC_HACTIVE) != 0) {

```

```

2174         DTRACE_PROBE1(sv_lyr_strategy_err_hactive, struct buf *, bp);
2176         cmn_err(CE_WARN,
2177             "!sv: allocated active handle (bufh %p, flags %x)",
2178             (void *)bufh, bufh->sb_flag);
2180         bioerror(bp, ENXIO);
2181         goto done;
2182     }
2184     fba_req = FBA_LEN(bp->b_bcount);
2185     if (fba_req + bp->b_lblkno > (diskaddr_t)svp->sv_nblocks)
2186         fba_req = (nsc_size_t)(svp->sv_nblocks - bp->b_lblkno);
2188     rw = (bp->b_flags & B_READ) ? NSC_READ : NSC_WRITE;
2190     bp_mapin(bp);
2192     bp->b_resid = bp->b_bcount;
2193     buf_addr = bp->b_un.b_addr;
2194     fba_off = 0;
2196     /*
2197     * fba_req - requested size of transfer in FBAs after
2198     * truncation to device extent, and allowing for
2199     * possible non-FBA bounded final chunk.
2200     * fba_off - offset of start of chunk from start of bp in FBAs.
2201     * fba_len - size of this chunk in FBAs.
2202     */
2204     loop:
2205     fba_len = min(fba_req, svp->sv_maxfbas);
2206     hndl = bufh;
2208     DTRACE_PROBE4(sv_dbg_alloch_start,
2209         sv_dev_t *, svp,
2210         uint64_t, (uint64_t)(bp->b_lblkno + fba_off),
2211         uint64_t, (uint64_t)fba_len,
2212         int, rw);
2214     rc = nsc_alloc_buf(svp->sv_fd, (nsc_off_t)(bp->b_lblkno + fba_off),
2215         fba_len, rw, &hndl);
2217     DTRACE_PROBE1(sv_dbg_alloch_end, sv_dev_t *, svp);
2219     if (rc > 0) {
2220         DTRACE_PROBE1(sv_lyr_strategy_err_alloc, struct buf *, bp);
2221         bioerror(bp, rc);
2222         if (hndl != bufh)
2223             (void) nsc_free_buf(hndl);
2224         hndl = NULL;
2225         goto done;
2226     }
2228     tocopy = min(FBA_SIZE(fba_len), bp->b_resid);
2229     v = hndl->sb_vec;
2231     if (rw == NSC_WRITE && FBA_OFF(tocopy) != 0) {
2232         /*
2233         * Not overwriting all of the last FBA, so read in the
2234         * old contents now before we overwrite it with the new
2235         * data.
2236         */
2238         DTRACE_PROBE2(sv_dbg_read_start, sv_dev_t *, svp,
2239             uint64_t, (uint64_t)(hndl->sb_pos + hndl->sb_len - 1));

```



```

2241         rc = nsc_read(hndl, (hndl->sb_pos + hndl->sb_len - 1), 1, 0);
2242         if (rc > 0) {
2243             bioerror(bp, rc);
2244             goto done;
2245         }
2247         DTRACE_PROBE1(sv_dbg_read_end, sv_dev_t *, svp);
2248     }
2250     DTRACE_PROBE1(sv_dbg_bcopy_start, sv_dev_t *, svp);
2252     while (tocopy > 0) {
2253         nbytes = min(tocopy, (nsc_size_t)v->sv_len);
2255         if (bp->b_flags & B_READ)
2256             (void) bcopy(v->sv_addr, buf_addr, nbytes);
2257         else
2258             (void) bcopy(buf_addr, v->sv_addr, nbytes);
2260         bp->b_resid -= nbytes;
2261         buf_addr += nbytes;
2262         tocopy -= nbytes;
2263         v++;
2264     }
2266     DTRACE_PROBE1(sv_dbg_bcopy_end, sv_dev_t *, svp);
2268     if ((bp->b_flags & B_READ) == 0) {
2269         DTRACE_PROBE3(sv_dbg_write_start, sv_dev_t *, svp,
2270             uint64_t, (uint64_t)hndl->sb_pos,
2271             uint64_t, (uint64_t)hndl->sb_len);
2273         rc = nsc_write(hndl, hndl->sb_pos, hndl->sb_len, 0);
2275         DTRACE_PROBE1(sv_dbg_write_end, sv_dev_t *, svp);
2277         if (rc > 0) {
2278             bioerror(bp, rc);
2279             goto done;
2280         }
2281     }
2283     /*
2284     * Adjust FBA offset and requested (ie. remaining) length,
2285     * loop if more data to transfer.
2286     */
2288     fba_off += fba_len;
2289     fba_req -= fba_len;
2291     if (fba_req > 0) {
2292         DTRACE_PROBE1(sv_dbg_freeb_start, sv_dev_t *, svp);
2294         rc = nsc_free_buf(hndl);
2296         DTRACE_PROBE1(sv_dbg_freeb_end, sv_dev_t *, svp);
2298         if (rc > 0) {
2299             DTRACE_PROBE1(sv_lyr_strategy_err_free,
2300                 struct buf *, bp);
2301             bioerror(bp, rc);
2302         }
2304         hndl = NULL;

```

```

2306         if (rc <= 0)
2307             goto loop;
2308     }
2310 done:
2311     if (hndl != NULL) {
2312         DTRACE_PROBE1(sv_dbg_freeb_start, sv_dev_t *, svp);
2314         rc = nsc_free_buf(hndl);
2316         DTRACE_PROBE1(sv_dbg_freeb_end, sv_dev_t *, svp);
2318         if (rc > 0) {
2319             DTRACE_PROBE1(sv_lyr_strategy_err_free,
2320                 struct buf *, bp);
2321             bioerror(bp, rc);
2322         }
2324         hndl = NULL;
2325     }
2327     if (bufh)
2328         (void) nsc_free_handle(bufh);
2330     DTRACE_PROBE1(sv_dbg_rlse_start, sv_dev_t *, svp);
2332     nsc_release(svp->sv_fd);
2334     DTRACE_PROBE1(sv_dbg_rlse_end, sv_dev_t *, svp);
2336 out:
2337     if (sv_debug > 5) {
2338         cmn_err(CE_CONT,
2339             "! sv_lyr_strategy: bp %p, bufh %p, bp->b_error %d\n",
2340             (void *)bp, (void *)bufh, bp->b_error);
2341     }
2343     DTRACE_PROBE2(sv_lyr_strategy_end, struct buf *, bp, int, bp->b_error);
2345     rw_exit(&svp->sv_lock);
2346     biodone(bp);
2347 }
2350 static void
2351 sv_async_strategy(blind_t arg)
2352 {
2353     struct buf *bp = (struct buf *)arg;
2354     _sv_lyr_strategy(bp);
2355 }
2358 static int
2359 sv_lyr_strategy(struct buf *bp)
2360 {
2361     nsthread_t *tp;
2362     int nlive;
2364     /*
2365     * If B_ASYNC was part of the DDI we could use it as a hint to
2366     * not create a thread for synchronous i/o.
2367     */
2368     if (sv_dev_to_sv(bp->b_edev, NULL) == NULL) {
2369         /* not sv enabled - just pass through */
2370         DTRACE_PROBE1(sv_lyr_strategy_notsv, struct buf *, bp);
2371         _sv_lyr_strategy(bp);

```

```

2372         return (0);
2373     }
2375     if (sv_debug > 4) {
2376         cmn_err(CE_CONT, "!sv_lyr_strategy: nthread %d nlive %d\n",
2377             nst_nthread(sv_tset), nst_nlive(sv_tset));
2378     }
2380     /*
2381     * If there are only guard devices enabled there
2382     * won't be a threadset, so don't try and use it.
2383     */
2384     tp = NULL;
2385     if (sv_tset != NULL) {
2386         tp = nst_create(sv_tset, sv_async_strategy, (blind_t)bp, 0);
2387     }
2389     if (tp == NULL) {
2390         /*
2391         * out of threads, so fall back to synchronous io.
2392         */
2393         if (sv_debug > 0) {
2394             cmn_err(CE_CONT,
2395                 "!sv_lyr_strategy: thread alloc failed\n");
2396         }
2398         DTRACE_PROBE1(sv_lyr_strategy_no_thread,
2399             struct buf *, bp);
2401         _sv_lyr_strategy(bp);
2402         sv_no_threads++;
2403     } else {
2404         nlive = nst_nlive(sv_tset);
2405         if (nlive > sv_max_nlive) {
2406             if (sv_debug > 0) {
2407                 cmn_err(CE_CONT,
2408                     "!sv_lyr_strategy: "
2409                     "new max nlive %d (nthread %d)\n",
2410                     nlive, nst_nthread(sv_tset));
2411             }
2413             sv_max_nlive = nlive;
2414         }
2415     }
2417     return (0);
2418 }

78 #ifndef offsetof
79 #define offsetof(s, m) ((size_t)&((s *)0)->m)
80 #endif

2420 /*
2421 * re-write the size of the current partition
2422 */
2423 static int
2424 sv_fix_dkiocgvtoc(const intptr_t arg, const int mode, sv_dev_t *svp)
2425 {
2426     size_t offset;
2427     int ilp32;
2428     int pnum;
2429     int rc;
2431     ilp32 = (ddi_model_convert_from((mode & FMODELS)) == DDI_MODEL_ILP32);

```

```

2433     rc = nskern_partition(svp->sv_dev, &pnum);
2434     if (rc != 0) {
2435         return (rc);
2436     }
2438     if (pnum < 0 || pnum >= V_NUMPAR) {
2439         cmn_err(CE_WARN,
2440             "!sv_gvtoc: unable to determine partition number "
2441             "for dev %lx", svp->sv_dev);
2442         return (EINVAL);
2443     }
2445     if (ilp32) {
2446         int32_t p_size;
2448 #ifdef _SunOS_5_6
2449         offset = offsetof(struct vtoc, v_part);
2450         offset += sizeof (struct partition) * pnum;
2451         offset += offsetof(struct partition, p_size);
2452 #else
2453         offset = offsetof(struct vtoc32, v_part);
2454         offset += sizeof (struct partition32) * pnum;
2455         offset += offsetof(struct partition32, p_size);
2456 #endif
2458         p_size = (int32_t)svp->sv_nblocks;
2459         if (p_size == 0) {
2460             if (sv_reserve(svp->sv_fd,
2461                 NSC_MULTI|NSC_PCATCH) == 0) {
2462                 p_size = (int32_t)svp->sv_nblocks;
2463                 nsc_release(svp->sv_fd);
2464             } else {
2465                 rc = EINTR;
2466             }
2467         }
2469         if ((rc == 0) && ddi_copyout(&p_size, (void *) (arg + offset),
2470             sizeof (p_size), mode) != 0) {
2471             rc = EFAULT;
2472         }
2473     } else {
2474         long p_size;
2476         offset = offsetof(struct vtoc, v_part);
2477         offset += sizeof (struct partition) * pnum;
2478         offset += offsetof(struct partition, p_size);
2480         p_size = (long)svp->sv_nblocks;
2481         if (p_size == 0) {
2482             if (sv_reserve(svp->sv_fd,
2483                 NSC_MULTI|NSC_PCATCH) == 0) {
2484                 p_size = (long)svp->sv_nblocks;
2485                 nsc_release(svp->sv_fd);
2486             } else {
2487                 rc = EINTR;
2488             }
2489         }
2491         if ((rc == 0) && ddi_copyout(&p_size, (void *) (arg + offset),
2492             sizeof (p_size), mode) != 0) {
2493             rc = EFAULT;
2494         }
2495     }
2497     return (rc);
2498 }

```

unchanged_portion_omitted

new/usr/src/uts/common/inet/ipf/netinet/ip_fil.h

1

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_____

```
615 #define fr_caddr      fr_dun.fru_caddr
616 #define fr_data       fr_dun.fru_data
617 #define fr_dfunc      fr_dun.fru_func
618 #define fr_ipf        fr_dun.fru_ipf
619 #define fr_ip         fr_ipf->fri_ip
620 #define fr_mip        fr_ipf->fri_mip
621 #define fr_icmpm      fr_ipf->fri_icmpm
622 #define fr_icmp       fr_ipf->fri_icmp
623 #define fr_tuc        fr_ipf->fri_tuc
624 #define fr_satype     fr_ipf->fri_satype
625 #define fr_datatype   fr_ipf->fri_datatype
626 #define fr_sifpidx    fr_ipf->fri_sifpidx
627 #define fr_difpidx   fr_ipf->fri_difpidx
628 #define fr_proto      fr_ip.fi_p
629 #define fr_mproto     fr_mip.fi_p
630 #define fr_ttl        fr_ip.fi_ttl
631 #define fr_mttl       fr_mip.fi_ttl
632 #define fr_tos        fr_ip.fi_tos
633 #define fr_mtos       fr_mip.fi_tos
634 #define fr_tcpfm      fr_tuc.ftu_tcpfm
635 #define fr_tcpf       fr_tuc.ftu_tcpf
636 #define fr_scmp       fr_tuc.ftu_scmp
637 #define fr_dcmp       fr_tuc.ftu_dcmp
638 #define fr_dport      fr_tuc.ftu_dport
639 #define fr_sport      fr_tuc.ftu_sport
640 #define fr_stop       fr_tuc.ftu_stop
641 #define fr_dtop       fr_tuc.ftu_dtop
642 #define fr_dst        fr_ip.fi_dst.in4
643 #define fr_daddr      fr_ip.fi_dst.in4.s_addr
644 #define fr_src        fr_ip.fi_src.in4
645 #define fr_saddr      fr_ip.fi_src.in4.s_addr
646 #define fr_dmask      fr_mip.fi_dst.in4
647 #define fr_dmasks     fr_mip.fi_dst.in4.s_addr
648 #define fr_smask      fr_mip.fi_src.in4
649 #define fr_smasks     fr_mip.fi_src.in4.s_addr
650 #define fr_dstnum     fr_ip.fi_dstnum
651 #define fr_srcnum     fr_ip.fi_srcnum
652 #define fr_dsttype    fr_ip.fi_dsttype
653 #define fr_srcctype   fr_ip.fi_srcctype
654 #define fr_dstptr     fr_mip.fi_dstptr
655 #define fr_srcptr     fr_mip.fi_srcptr
656 #define fr_dstfunc    fr_mip.fi_dstfunc
657 #define fr_srcfunc    fr_mip.fi_srcfunc
658 #define fr_optbits    fr_ip.fi_optmsk
659 #define fr_optmask    fr_mip.fi_optmsk
660 #define fr_secbits    fr_ip.fi_secmsk
661 #define fr_secmask    fr_mip.fi_secmsk
662 #define fr_authbits   fr_ip.fi_auth
663 #define fr_authmask   fr_mip.fi_auth
664 #define fr_flx        fr_ip.fi_flx
665 #define fr_mflx       fr_mip.fi_flx
666 #define fr_ifname     fr_ifnames[0]
667 #define fr_oifname    fr_ifnames[2]
668 #define fr_ifa        fr_ifas[0]
669 #define fr_oifa       fr_ifas[2]
670 #define fr_tif        fr_tifs[0]
671 #define fr_rif        fr_tifs[1]

673 #define FR_NOLOGTAG    0
```

new/usr/src/uts/common/inet/ipf/netinet/ip_fil.h

2

```
675 #ifndef offsetof
676 #define offsetof(t,m)  ((size_t)((t *)0->m))
677 #endif
675 #define FR_CMPSIZ      (sizeof(struct frentry) - \
676                        offsetof(struct frentry, fr_func))

678 /*
679  * fr_type
680  */
681 #define FR_T_NONE      0
682 #define FR_T_IPF       1 /* IPF structures */
683 #define FR_T_BPFOPC    2 /* BPF opcode */
684 #define FR_T_CALLFUNC  3 /* callout to function in fr_func only */
685 #define FR_T_COMPIPF   4 /* compiled C code */
686 #define FR_T_BUILTIN   0x80000000 /* rule is in kernel space */

688 /*
689  * fr_flags
690  */
691 #define FR_CALL        0x00000 /* call rule */
692 #define FR_BLOCK       0x00001 /* do not allow packet to pass */
693 #define FR_PASS        0x00002 /* allow packet to pass */
694 #define FR_AUTH        0x00003 /* use authentication */
695 #define FR_PREAUTH     0x00004 /* require preauthentication */
696 #define FR_ACCOUNT     0x00005 /* Accounting rule */
697 #define FR_SKIP        0x00006 /* skip rule */
698 #define FR_DIVERT      0x00007 /* divert rule */
699 #define FR_CMDMASK     0x0000f
700 #define FR_LOG         0x00010 /* Log */
701 #define FR_LOGB        0x00011 /* Log-fail */
702 #define FR_LOGP        0x00012 /* Log-pass */
703 #define FR_LOGMASK     (FR_LOG|FR_CMDMASK)
704 #define FR_CALLNOW     0x00020 /* call another function (fr_func) if matches */
705 #define FR_NOTSRCIP    0x00040
706 #define FR_NOTDSTIP    0x00080
707 #define FR_QUICK       0x00100 /* match & stop processing list */
708 #define FR_KEEPPRAG    0x00200 /* keep fragment information */
709 #define FR_KEEPPSTATE  0x00400 /* keep 'connection' state information */
710 #define FR_FASTROUTE   0x00800 /* bypass normal routing */
711 #define FR_RETRST      0x01000 /* Return TCP RST packet - reset connection */
712 #define FR_RETICMP     0x02000 /* Return ICMP unreachable packet */
713 #define FR_FAKEICMP    0x03000 /* Return ICMP unreachable with fake source */
714 #define FR_OUTQUE      0x04000 /* outgoing packets */
715 #define FR_INQUE       0x08000 /* ingoing packets */
716 #define FR_LOGBODY     0x10000 /* Log the body */
717 #define FR_LOGFIRST    0x20000 /* Log the first byte if state held */
718 #define FR_LOGORBLOCK  0x40000 /* block the packet if it can't be logged */
719 #define FR_DUP         0x80000 /* duplicate packet */
720 #define FR_FRSTRICT    0x100000 /* strict frag. cache */
721 #define FR_STRICT      0x200000 /* strict keep state */
722 #define FR_NEWISN      0x400000 /* new ISN for outgoing TCP */
723 #define FR_NOICMPERR   0x800000 /* do not match ICMP errors in state */
724 #define FR_STATESYNC   0x1000000 /* synchronize state to slave */
725 #define FR_NOMATCH     0x8000000 /* no match occurred */
726 /*
727  */
727 /*
728  */
728 /*
729  */
729 /*
730 #define FR_COPIED      0x40000000 /* copied from user space */
731 #define FR_INACTIVE    0x80000000 /* only used when flush'ing rules */

733 #define FR_RETICMP     (FR_RETICMP|FR_RETRST|FR_FAKEICMP)
734 #define FR_ISBLOCK(x)  (((x) & FR_CMDMASK) == FR_BLOCK)
735 #define FR_ISPASS(x)   (((x) & FR_CMDMASK) == FR_PASS)
736 #define FR_ISAUTH(x)   (((x) & FR_CMDMASK) == FR_AUTH)
```

```
737 #define FR_ISPREAUTH(x) (((x) & FR_CMDMASK) == FR_PREAUTH)
738 #define FR_ISACCOUNT(x) (((x) & FR_CMDMASK) == FR_ACCOUNT)
739 #define FR_ISSKIP(x) (((x) & FR_CMDMASK) == FR_SKIP)
740 #define FR_ISNOMATCH(x) ((x) & FR_NOMATCH)
741 #define FR_INOUT (FR_INQUE|FR_OUTQUE)

743 /*
744  * recognized flags for SIOCGETFF and SIOCSETFF, and get put in fr_flags
745  */
746 #define FF_LOGPASS 0x10000000
747 #define FF_LOGBLOCK 0x20000000
748 #define FF_LOGNOMATCH 0x40000000
749 #define FF_LOGGING (FF_LOGPASS|FF_LOGBLOCK|FF_LOGNOMATCH)
750 #define FF_BLOCKNONIP 0x80000000 /* Solaris2 Only */

753 /*
754  * Structure that passes information on what/how to flush to the kernel.
755  */
756 typedef struct ipfflush {
757     int ipflu_how;
758     int ipflu_arg;
759 } 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
*****
1 /*
2  * drmP.h -- Private header for Direct Rendering Manager -*- linux-c -*-
3  * Created: Mon Jan 4 10:05:05 1999 by faith@precisioninsight.com
4  */
5 /*
6  * Copyright 1999 Precision Insight, Inc., Cedar Park, Texas.
7  * Copyright 2000 VA Linux Systems, Inc., Sunnyvale, California.
8  * Copyright (c) 2009, Intel Corporation.
9  * All rights reserved.
10 *
11 * Permission is hereby granted, free of charge, to any person obtaining a
12 * copy of this software and associated documentation files (the "Software"),
13 * to deal in the Software without restriction, including without limitation
14 * the rights to use, copy, modify, merge, publish, distribute, sublicense,
15 * and/or sell copies of the Software, and to permit persons to whom the
16 * Software is furnished to do so, subject to the following conditions:
17 *
18 * The above copyright notice and this permission notice (including the next
19 * paragraph) shall be included in all copies or substantial portions of the
20 * Software.
21 *
22 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
23 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
24 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
25 * VA LINUX SYSTEMS AND/OR ITS SUPPLIERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
26 * OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
27 * ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
28 * OTHER DEALINGS IN THE SOFTWARE.
29 *
30 * Authors:
31 *   Rickard E. (Rik) Faith <faith@valinux.com>
32 *   Gareth Hughes <gareth@valinux.com>
33 *
34 */

36 /*
37  * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
38  * Use is subject to license terms.
39  */

41 #ifndef _DRMP_H
42 #define _DRMP_H

44 #include <sys/sysmacros.h>
45 #include <sys/types.h>
46 #include <sys/conf.h>
47 #include <sys/modctl.h>
48 #include <sys/stat.h>
49 #include <sys/file.h>
50 #include <sys/cmn_err.h>
51 #include <sys/varargs.h>
52 #include <sys/pci.h>
53 #include <sys/ddi.h>
54 #include <sys/sunddi.h>
55 #include <sys/sunldi.h>
56 #include <sys/pmem.h>
57 #include <sys/agpgart.h>
58 #include <sys/time.h>
59 #include <sys/sysmacros.h>
60 #endif /* ! codereview */
61 #include "drm_atomic.h"

```

```

62 #include "drm.h"
63 #include "queue.h"
64 #include "drm_linux_list.h"

66 #ifndef __inline__
67 #define __inline__ inline
68 #endif

70 #if !defined(__FUNCTION__)
71 #if defined(C99)
72 #define __FUNCTION__ __func__
73 #else
74 #define __FUNCTION__ " "
75 #endif
76 #endif

78 /* DRM space units */
79 #define DRM_PAGE_SHIFT          PAGESHIFT
80 #define DRM_PAGE_SIZE          (1 << DRM_PAGE_SHIFT)
81 #define DRM_PAGE_OFFSET        (DRM_PAGE_SIZE - 1)
82 #define DRM_PAGE_MASK          ~(DRM_PAGE_SIZE - 1)
83 #define DRM_MB2PAGES(x)        ((x) << 8)
84 #define DRM_PAGES2BYTES(x)     ((x) << DRM_PAGE_SHIFT)
85 #define DRM_BYTES2PAGES(x)     ((x) >> DRM_PAGE_SHIFT)
86 #define DRM_PAGES2KB(x)        ((x) << 2)
87 #define DRM_ALIGNED(offset)    (((offset) & DRM_PAGE_OFFSET) == 0)

89 #define PAGE_SHIFT              DRM_PAGE_SHIFT
90 #define PAGE_SIZE               DRM_PAGE_SIZE

92 #define DRM_MAX_INSTANCES       8
93 #define DRM_DEVMODE             "drm"
94 #define DRM_UNOPENED           0
95 #define DRM_OPENED              1

97 #define DRM_HASH_SIZE           16 /* Size of key hash table */
98 #define DRM_KERNEL_CONTEXT      0 /* Change drm_resctx if changed */
99 #define DRM_RESERVED_CONTEXTS   1 /* Change drm_resctx if changed */

101 #define DRM_MEM_DMA             0
102 #define DRM_MEM_SAREA           1
103 #define DRM_MEM_DRIVER          2
104 #define DRM_MEM_MAGIC           3
105 #define DRM_MEM_IOCTLs         4
106 #define DRM_MEM_MAPS            5
107 #define DRM_MEM_BUFS            6
108 #define DRM_MEM_SEGS            7
109 #define DRM_MEM_PAGES           8
110 #define DRM_MEM_FILES           9
111 #define DRM_MEM_QUEUES          10
112 #define DRM_MEM_CMDS            11
113 #define DRM_MEM_MAPPINGS        12
114 #define DRM_MEM_BUFLISTS        13
115 #define DRM_MEM_DRMLISTS        14
116 #define DRM_MEM_TOTALDRM        15
117 #define DRM_MEM_BOUNDDRM        16
118 #define DRM_MEM_CTXBITMAP       17
119 #define DRM_MEM_STUB            18
120 #define DRM_MEM_SGLISTS         19
121 #define DRM_MEM_AGPLISTS        20
122 #define DRM_MEM_CTXLIST         21
123 #define DRM_MEM_MM              22
124 #define DRM_MEM_HASHTAB         23
125 #define DRM_MEM_OBJECTS         24

127 #define DRM_MAX_CTXBITMAP (PAGE_SIZE * 8)

```

```

128 #define DRM_MAP_HASH_OFFSET 0x10000000
129 #define DRM_MAP_HASH_ORDER 12
130 #define DRM_OBJECT_HASH_ORDER 12
131 #define DRM_FILE_PAGE_OFFSET_START ((0xFFFFFFFFFUL >> PAGE_SHIFT) + 1)
132 #define DRM_FILE_PAGE_OFFSET_SIZE ((0xFFFFFFFFFUL >> PAGE_SHIFT) * 16)
133 #define DRM_MM_INIT_MAX_PAGES 256

136 /* Internal types and structures */
137 #define DRM_ARRAY_SIZE(x) (sizeof(x) / sizeof(x[0]))
138 #define DRM_MIN(a, b) ((a) < (b) ? (a) : (b))
139 #define DRM_MAX(a, b) ((a) > (b) ? (a) : (b))

141 #define DRM_IF_VERSION(maj, min) (maj << 16 | min)

143 #define __OS_HAS_AGP 1

145 #define DRM_DEV_MOD (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP)
146 #define DRM_DEV_UID 0
147 #define DRM_DEV_GID 0

149 #define DRM_CURRENTPID ddi_get_pid()
150 #define DRM_SPINLOCK(l) mutex_enter(l)
151 #define DRM_SPINUNLOCK(u) mutex_exit(u)
152 #define DRM_SPINLOCK_ASSERT(l)
153 #define DRM_LOCK() mutex_enter(&dev->dev_lock)
154 #define DRM_UNLOCK() mutex_exit(&dev->dev_lock)
155 #define DRM_LOCK_OWNED() ASSERT(mutex_owned(&dev->dev_lock))
156 #define spin_lock_irqsave(l, flag) mutex_enter(l)
157 #define spin_unlock_irqrestore(u, flag) mutex_exit(u)
158 #define spin_lock(l) mutex_enter(l)
159 #define spin_unlock(u) mutex_exit(u)

162 #define DRM_UDELAY(sec) delay(drv_usectohz(sec * 1000))
163 #define DRM_MEMORYBARRIER()

165 typedef struct drm_file drm_file_t;
166 typedef struct drm_device drm_device_t;
167 typedef struct drm_driver_info drm_driver_t;

169 #define DRM_DEVICE drm_device_t *dev = dev1
170 #define DRM_IOCTL_ARGS \
171     drm_device_t *dev1, intptr_t data, drm_file_t *fpriv, int mode

173 #define DRM_COPYFROM_WITH_RETURN(dest, src, size) \
174     if (ddi_copyin((src), (dest), (size), 0)) { \
175         DRM_ERROR("%s: copy from user failed", __func__); \
176         return (EFAULT); \
177     }

179 #define DRM_COPYTO_WITH_RETURN(dest, src, size) \
180     if (ddi_copyout((src), (dest), (size), 0)) { \
181         DRM_ERROR("%s: copy to user failed", __func__); \
182         return (EFAULT); \
183     }

185 #define DRM_COPY_FROM_USER(dest, src, size) \
186     ddi_copyin((src), (dest), (size), 0) /* flag for src */

188 #define DRM_COPY_TO_USER(dest, src, size) \
189     ddi_copyout((src), (dest), (size), 0) /* flags for dest */

191 #define DRM_COPY_FROM_USER_UNCHECKED(arg1, arg2, arg3) \
192     ddi_copyin((arg2), (arg1), (arg3), 0)

```

```

194 #define DRM_COPY_TO_USER_UNCHECKED(arg1, arg2, arg3) \
195     ddi_copyout((arg2), arg1, arg3, 0)

197 #define DRM_READ8(map, offset) \
198     *(volatile uint8_t *)((uintptr_t)((map)->dev_addr) + (offset))
199 #define DRM_READ16(map, offset) \
200     *(volatile uint16_t *)((uintptr_t)((map)->dev_addr) + (offset))
201 #define DRM_READ32(map, offset) \
202     *(volatile uint32_t *)((uintptr_t)((map)->dev_addr) + (offset))
203 #define DRM_WRITE8(map, offset, val) \
204     *(volatile uint8_t *)((uintptr_t)((map)->dev_addr) + (offset)) = (val)
205 #define DRM_WRITE16(map, offset, val) \
206     *(volatile uint16_t *)((uintptr_t)((map)->dev_addr) + (offset)) = (val)
207 #define DRM_WRITE32(map, offset, val) \
208     *(volatile uint32_t *)((uintptr_t)((map)->dev_addr) + (offset)) = (val)

210 typedef struct drm_wait_queue {
211     kcondvar_t cv;
212     kmutex_t lock;
213 } wait_queue_head_t;

215 #define DRM_INIT_WAITQUEUE(q, pri) \
216 { \
217     mutex_init(&(q)->lock, NULL, MUTEX_DRIVER, pri); \
218     cv_init(&(q)->cv, NULL, CV_DRIVER, NULL); \
219 }

221 #define DRM_FINI_WAITQUEUE(q) \
222 { \
223     mutex_destroy(&(q)->lock); \
224     cv_destroy(&(q)->cv); \
225 }

227 #define DRM_WAKEUP(q) \
228 { \
229     mutex_enter(&(q)->lock); \
230     cv_broadcast(&(q)->cv); \
231     mutex_exit(&(q)->lock); \
232 }

234 #define jiffies ddi_get_lbolt()

236 #define DRM_WAIT_ON(ret, q, timeout, condition) \
237     mutex_enter(&(q)->lock); \
238     while (!(condition)) { \
239         ret = cv_reltimedwait_sig(&(q)->cv, &(q)->lock, timeout, \
240             TR_CLOCK_TICK); \
241         if (ret == -1) { \
242             ret = EBUSY; \
243             break; \
244         } else if (ret == 0) { \
245             ret = EINTR; \
246             break; \
247         } else { \
248             ret = 0; \
249         } \
250     } \
251     mutex_exit(&(q)->lock);

253 #define DRM_WAIT(ret, q, condition) \
254     mutex_enter(&(q)->lock); \
255     if (!(condition)) { \
256         ret = cv_timedwait_sig(&(q)->cv, &(q)->lock, jiffies + 30 * DRM_HZ); \
257         if (ret == -1) { \
258             /* gfx maybe hang */ \
259             if (!(condition)) \

```

```

260         ret = -2;
261     } else {
262         ret = 0;
263     }
264 } \
265 mutex_exit(&(q)->lock);

268 #define DRM_GETSAREA() \
269 { \
270     drm_local_map_t *map; \
271     DRM_SPINLOCK_ASSERT(&dev->dev_lock); \
272     TAILQ_FOREACH(map, &dev->maplist, link) { \
273         if (map->type == _DRM_SHM && \
274             map->flags & _DRM_CONTAINS_LOCK) { \
275             dev_priv->sarea = map; \
276             break; \
277         } \
278     } \
279 }

281 #define LOCK_TEST_WITH_RETURN(dev, fpriv) \
282 if (!DRM_LOCK_IS_HELD(dev->lock.hw_lock->lock) || \
283     dev->lock.filp != fpriv) { \
284     DRM_DEBUG("%s called without lock held", __func__); \
285     return (EINVAL); \
286 }

288 #define DRM_IRQ_ARGS    caddr_t arg
289 #define IRQ_HANDLED    DDI_INTR_CLAIMED
290 #define IRQ_NONE        DDI_INTR_UNCLAIMED

292 enum {
293     DRM_IS_NOT_AGP,
294     DRM_IS_AGP,
295     DRM_MIGHT_BE_AGP
296 };

298 /* Capabilities taken from src/sys/dev/pci/pci.h. */
299 #ifndef PCIY_AGP
300 #define PCIY_AGP        0x02
301 #endif

303 #ifndef PCIY_EXPRESS
304 #define PCIY_EXPRESS    0x10
305 #endif

307 #define PAGE_ALIGN(addr)    (((addr) + DRM_PAGE_SIZE - 1) & DRM_PAGE_MASK)
308 #define DRM_SUSER(p)        (crgetsgid(p) == 0 || crgetsuid(p) == 0)

310 #define DRM_GEM_OBJIDR_HASHNODE 1024
311 #define idr_list_for_each(entry, head) \
312 for (int key = 0; key < DRM_GEM_OBJIDR_HASHNODE; key++) \
313     list_for_each(entry, &(head)->next[key])

315 /*
316  * wait for 400 milliseconds
317  */
318 #define DRM_HZ            drv_usecstohz(400000)

320 typedef unsigned long dma_addr_t;
321 typedef uint64_t        u64;
322 typedef uint32_t        u32;
323 typedef uint16_t        u16;
324 typedef uint8_t         u8;
325 typedef uint_t          irqreturn_t;

```

```

327 #define DRM_SUPPORT        1
328 #define DRM_UN SUPPORT      0

330 #define __OS_HAS_AGP        1

332 typedef struct drm_pci_id_list
333 {
334     int vendor;
335     int device;
336     long driver_private;
337     char *name;
338 } drm_pci_id_list_t;

340 #define DRM_AUTH            0x1
341 #define DRM_MASTER          0x2
342 #define DRM_ROOT_ONLY      0x4
343 typedef int drm_ioctl_t(DRM_IOCTL_ARGS);
344 typedef struct drm_ioctl_desc {
345     int (*func)(DRM_IOCTL_ARGS);
346     int flags;
347 } drm_ioctl_desc_t;

349 typedef struct drm_magic_entry {
350     drm_magic_t magic;
351     struct drm_file *priv;
352     struct drm_magic_entry *next;
353 } drm_magic_entry_t;

355 typedef struct drm_magic_head {
356     struct drm_magic_entry *head;
357     struct drm_magic_entry *tail;
358 } drm_magic_head_t;

360 typedef struct drm_buf {
361     int idx; /* Index into master buflist */
362     int total; /* Buffer size */
363     int order; /* log-base-2(total) */
364     int used; /* Amount of buffer in use (for DMA) */
365     unsigned long offset; /* Byte offset (used internally) */
366     void *address; /* Address of buffer */
367     unsigned long bus_address; /* Bus address of buffer */
368     struct drm_buf *next; /* Kernel-only: used for free list */
369     volatile int pending; /* On hardware DMA queue */
370     drm_file_t *filp;
371     /* Uniq. identifier of holding process */
372     int context; /* Kernel queue for this buffer */
373     enum {
374         DRM_LIST_NONE = 0,
375         DRM_LIST_FREE = 1,
376         DRM_LIST_WAIT = 2,
377         DRM_LIST PEND = 3,
378         DRM_LIST_PRIO = 4,
379         DRM_LIST RECLAIM = 5
380     } list; /* Which list we're on */

382     int dev_priv_size; /* Size of buffer private storage */
383     void *dev_private; /* Per-buffer private storage */
384 } drm_buf_t;

386 typedef struct drm_freelist {
387     int initialized; /* Freelist in use */
388     uint32_t count; /* Number of free buffers */
389     drm_buf_t *next; /* End pointer */
391     int low_mark; /* Low water mark */

```

```

392     int                high_mark;    /* High water mark          */
393 } drm_freelist_t;

395 typedef struct drm_buf_entry {
396     int                buf_size;
397     int                buf_count;
398     drm_buf_t         *buflist;
399     int                seg_count;
400     int                page_order;

402     uint32_t          *seglst;
403     unsigned long     *seglst_bus;

405     drm_freelist_t    freelist;
406 } drm_buf_entry_t;

408 typedef TAILQ_HEAD(drm_file_list, drm_file) drm_file_list_t;

410 /* BEGIN CSTYLED */
411 typedef struct drm_local_map {
412     unsigned long     offset; /* Physical address (0 for SAREA) */
413     unsigned long     size;   /* Physical size (bytes)          */
414     drm_map_type_t    type;   /* Type of memory mapped         */
415     drm_map_flags_t   flags;  /* Flags                          */
416     void              *handle; /* User-space: "Handle" to pass to mmap */
417                               /* Kernel-space: kernel-virtual address */
418     int               mtrr;   /* Boolean: MTRR used            */
419                               /* Private data                   */
420     int               rid;    /* PCI resource ID for bus_space */
421     int               kernel_owned; /* Boolean: 1= initmapped, 0= addmapped */
422     caddr_t           dev_addr; /* base device address          */
423     ddi_acc_handle_t  dev_handle; /* The data access handle      */
424     ddi_umem_cookie_t drm_umem_cookie; /* For SAREA alloc and free */
425     TAILQ_ENTRY(drm_local_map) link;
426 } drm_local_map_t;
427 /* END CSTYLED */

429 /*
430 * This structure defines the drm_mm memory object, which will be used by the
431 * DRM for its buffer objects.
432 */
433 struct drm_gem_object {
434     /* Reference count of this object */
435     atomic_t refcount;

437     /* Handle count of this object. Each handle also holds a reference */
438     atomic_t handlecount;

440     /* Related drm device */
441     struct drm_device *dev;

443     int flink;
444     /*
445      * Size of the object, in bytes. Immutable over the object's
446      * lifetime.
447      */
448     size_t size;

450     /*
451      * Global name for this object, starts at 1. 0 means unnamed.
452      * Access is covered by the object_name_lock in the related drm_device
453      */
454     int name;

456     /*
457      * Memory domains. These monitor which caches contain read/write data

```

```

458     * related to the object. When transitioning from one set of domains
459     * to another, the driver is called to ensure that caches are suitably
460     * flushed and invalidated
461     */
462     uint32_t read_domains;
463     uint32_t write_domain;

465     /*
466     * While validating an exec operation, the
467     * new read/write domain values are computed here.
468     * They will be transferred to the above values
469     * at the point that any cache flushing occurs
470     */
471     uint32_t pending_read_domains;
472     uint32_t pending_write_domain;

474     void *driver_private;

476     drm_local_map_t *map;
477     ddi_dma_handle_t dma_hdl;
478     ddi_acc_handle_t acc_hdl;
479     caddr_t kaddr;
480     size_t real_size; /* real size of memory */
481     pfn_t *pfnarray;
482 };

484 struct idr_list {
485     struct idr_list *next, *prev;
486     struct drm_gem_object *obj;
487     uint32_t handle;
488     caddr_t contain_ptr;
489 };

491 struct drm_file {
492     TAILQ_ENTRY(drm_file) link;
493     int authenticated;
494     int master;
495     int minor;
496     pid_t pid;
497     uid_t uid;
498     int refs;
499     drm_magic_t magic;
500     unsigned long ioctl_count;
501     void *driver_priv;
502     /* Mapping of mm object handles to object pointers. */
503     struct idr_list object_idr;
504     /* Lock for synchronization of access to object_idr. */
505     kmutex_t table_lock;

507     dev_t dev;
508     cred_t *credp;
509 };

511 typedef struct drm_lock_data {
512     drm_hw_lock_t *hw_lock; /* Hardware lock */
513     drm_file_t *filp;
514     /* Uniq. identifier of holding process */
515     kcondvar_t lock_cv; /* lock queue - SOLARIS Specific */
516     kmutex_t lock_mutex; /* lock - SOLARIS Specific */
517     unsigned long lock_time; /* Time of last lock in clock ticks */
518 } drm_lock_data_t;

520 /*
521 * This structure, in drm_device_t, is always initialized while the device
522 * is open. dev->dma_lock protects the incrementing of dev->buf_use, which
523 * when set marks that no further bufs may be allocated until device teardown

```



```

524 * occurs (when the last open of the device has closed). The high/low
525 * watermarks of bufs are only touched by the X Server, and thus not
526 * concurrently accessed, so no locking is needed.
527 */
528 typedef struct drm_device_dma {
529     drm_buf_entry_t bufs[DRM_MAX_ORDER+1];
530     int buf_count;
531     drm_buf_t **buflist; /* Vector of pointers info bufs */
532     int seg_count;
533     int page_count;
534     unsigned long *pagelist;
535     unsigned long byte_count;
536     enum {
537         _DRM_DMA_USE_AGP = 0x01,
538         _DRM_DMA_USE_SG = 0x02
539     } flags;
540 } drm_device_dma_t;

542 typedef struct drm_agp_mem {
543     void *handle;
544     unsigned long bound; /* address */
545     int pages;
546     caddr_t phys_addr;
547     struct drm_agp_mem *prev;
548     struct drm_agp_mem *next;
549 } drm_agp_mem_t;

551 typedef struct drm_agp_head {
552     agp_info_t agp_info;
553     const char *chipset;
554     drm_agp_mem_t *memory;
555     unsigned long mode;
556     int enabled;
557     int acquired;
558     unsigned long base;
559     int mtrr;
560     int cant_use_aperture;
561     unsigned long page_mask;
562     ldi_ident_t agpgart_li;
563     ldi_handle_t agpgart_lh;
564 } drm_agp_head_t;

567 typedef struct drm_dma_handle {
568     ddi_dma_handle_t dma_hdl;
569     ddi_acc_handle_t acc_hdl;
570     ddi_dma_cookie_t cookie;
571     uint_t cookie_num;
572     uintptr_t vaddr; /* virtual addr */
573     uintptr_t paddr; /* physical addr */
574     size_t real_sz; /* real size of memory */
575 } drm_dma_handle_t;

577 typedef struct drm_sg_mem {
578     unsigned long handle;
579     void *virtual;
580     int pages;
581     dma_addr_t *busaddr;
582     ddi_umem_cookie_t *umem_cookie;
583     drm_dma_handle_t *dmah_sg;
584     drm_dma_handle_t *dmah_gart; /* Handle to PCI memory */
585 } drm_sg_mem_t;

587 /*
588 * Generic memory manager structs
589 */

```

```

591 struct drm_mm_node {
592     struct list_head fl_entry;
593     struct list_head ml_entry;
594     int free;
595     unsigned long start;
596     unsigned long size;
597     struct drm_mm *mm;
598     void *private;
599 };

601 struct drm_mm {
602     struct list_head fl_entry;
603     struct list_head ml_entry;
604 };

606 typedef TAILQ_HEAD(drm_map_list, drm_local_map) drm_map_list_t;

608 typedef TAILQ_HEAD(drm_vbl_sig_list, drm_vbl_sig) drm_vbl_sig_list_t;
609 typedef struct drm_vbl_sig {
610     TAILQ_ENTRY(drm_vbl_sig) link;
611     unsigned int sequence;
612     int signo;
613     int pid;
614 } drm_vbl_sig_t;

617 /* used for clone device */
618 typedef TAILQ_HEAD(drm_cminor_list, drm_cminor) drm_cminor_list_t;
619 typedef struct drm_cminor {
620     TAILQ_ENTRY(drm_cminor) link;
621     drm_file_t *fpriv;
622     int minor;
623 } drm_cminor_t;

625 /* location of GART table */
626 #define DRM_ATI_GART_MAIN 1
627 #define DRM_ATI_GART_FB 2

629 typedef struct ati_pcigart_info {
630     int gart_table_location;
631     int is_pcie;
632     void *addr;
633     dma_addr_t bus_addr;
634     drm_local_map_t mapping;
635 } drm_ati_pcigart_info;

637 /* DRM device structure */
638 struct drm_device;
639 struct drm_driver_info {
640     int (*load)(struct drm_device *, unsigned long);
641     int (*firstopen)(struct drm_device *);
642     int (*open)(struct drm_device *, drm_file_t *);
643     void (*preclose)(struct drm_device *, drm_file_t *);
644     void (*postclose)(struct drm_device *, drm_file_t *);
645     void (*lastclose)(struct drm_device *);
646     int (*unload)(struct drm_device *);
647     void (*reclaim_buffers_locked)(struct drm_device *, drm_file_t *);
648     int (*presetup)(struct drm_device *);
649     int (*postsetup)(struct drm_device *);
650     int (*open_helper)(struct drm_device *, drm_file_t *);
651     void (*free_fil_priv)(struct drm_device *, drm_file_t *);
652     void (*release)(struct drm_device *, void *);
653     int (*dma_ioctl)(DRM_IOCTL_ARGS);
654     void (*dma_ready)(struct drm_device *);
655     int (*dma_quiescent)(struct drm_device *);

```

```

656 int (*dma_flush_block_and_flush)(struct drm_device *,
657     int, drm_lock_flags_t);
658 int (*dma_flush_unlock)(struct drm_device *, int,
659     drm_lock_flags_t);
660 int (*context_ctor)(struct drm_device *, int);
661 int (*context_dtor)(struct drm_device *, int);
662 int (*kernel_context_switch)(struct drm_device *, int, int);
663 int (*kernel_context_switch_unlock)(struct drm_device *);
664 int (*device_is_agp)(struct drm_device *);
665 int (*irq_preinstall)(struct drm_device *);
666 void (*irq_postinstall)(struct drm_device *);
667 void (*irq_uninstall)(struct drm_device *dev);
668 uint_t (*irq_handler)(DRM_IRQ_ARGS);
669 int (*vblank_wait)(struct drm_device *, unsigned int *);
670 int (*vblank_wait2)(struct drm_device *, unsigned int *);
671 /* added for intel minimized vblank */
672 u32 (*get_vblank_counter)(struct drm_device *dev, int crtc);
673 int (*enable_vblank)(struct drm_device *dev, int crtc);
674 void (*disable_vblank)(struct drm_device *dev, int crtc);

676 /*
677  * Driver-specific constructor for drm_gem_objects, to set up
678  * obj->driver_private.
679  *
680  * Returns 0 on success.
681  */
682 int (*gem_init_object)(struct drm_gem_object *obj);
683 void (*gem_free_object)(struct drm_gem_object *obj);

686 drm_ioctl_desc_t *driver_ioctls;
687 int max_driver_ioctl;

689 int buf_priv_size;
690 int driver_major;
691 int driver_minor;
692 int driver_patchlevel;
693 const char *driver_name; /* Simple driver name */
694 const char *driver_desc; /* Longer driver name */
695 const char *driver_date; /* Date of last major changes. */

697 unsigned use_agp :1;
698 unsigned require_agp :1;
699 unsigned use_sg :1;
700 unsigned use_dma :1;
701 unsigned use_pci_dma :1;
702 unsigned use_dma_queue :1;
703 unsigned use_irq :1;
704 unsigned use_vbl_irq :1;
705 unsigned use_vbl_irq2 :1;
706 unsigned use_mtrr :1;
707 unsigned use_gem;
708 };

710 /*
711  * hardware-specific code needs to initialize mutexes which
712  * can be used in interrupt context, so they need to know
713  * the interrupt priority. Interrupt cookie in drm_device
714  * structure is the intr_block field.
715  */
716 #define DRM_INTR_PRI(dev) \
717     DDI_INTR_PRI((dev)->intr_block)

719 struct drm_device {
720     drm_driver_t *driver;
721     drm_cminor_list_t minordevs;

```

```

722 dev_info_t *dip;
723 void *drm_handle;
724 int drm_supported;
725 const char *desc; /* current driver description */
726 kmutex_t *irq_mutex;
727 kcondvar_t *irq_cv;

729 ddi_iblock_cookie_t intr_block;
730 uint32_t pci_device; /* PCI device id */
731 uint32_t pci_vendor;
732 char *unique; /* Unique identifier: e.g., busid */
733 int unique_len; /* Length of unique field */
734 int if_version; /* Highest interface version set */
735 int flags; /* Flags to open(2) */

737 /* Locks */
738 kmutex_t vbl_lock; /* protects vblank operations */
739 kmutex_t dma_lock; /* protects dev->dma */
740 kmutex_t irq_lock; /* protects irq condition checks */
741 kmutex_t dev_lock; /* protects everything else */
742 drm_lock_data_t lock; /* Information on hardware lock */
743 kmutex_t struct_mutex; /* < For others */

745 /* Usage Counters */
746 int open_count; /* Outstanding files open */
747 int buf_use; /* Buffers in use -- cannot alloc */

749 /* Performance counters */
750 unsigned long counters;
751 drm_stat_type_t types[15];
752 uint32_t counts[15];

754 /* Authentication */
755 drm_file_list_t files;
756 drm_magic_head_t magiclist[DRM_HASH_SIZE];

758 /* Linked list of mappable regions. Protected by dev_lock */
759 drm_map_list_t maplist;

761 drm_local_map_t **context_sareas;
762 int max_context;

764 /* DMA queues (contexts) */
765 drm_device_dma_t *dma; /* Optional pointer for DMA support */

767 /* Context support */
768 int irq; /* Interrupt used by board */
769 int irq_enabled; /* True if the irq handler is enabled */
770 int pci_domain;
771 int pci_bus;
772 int pci_slot;
773 int pci_func;
774 atomic_t context_flag; /* Context swapping flag */
775 int last_context; /* Last current context */

777 /* Only used for Radeon */
778 atomic_t vbl_received;
779 atomic_t vbl_received2;

781 drm_vbl_sig_list_t vbl_sig_list;
782 drm_vbl_sig_list_t vbl_sig_list2;
783 /*
784  * At load time, disabling the vblank interrupt won't be allowed since
785  * old clients may not call the modeset ioctl and therefore misbehave.
786  * Once the modeset ioctl *has* been called though, we can safely
787  * disable them when unused.

```

```

788  */
789  int vblank_disable_allowed;

791  wait_queue_head_t vbl_queue; /* vbl wait channel */
792  /* vbl wait channel array */
793  wait_queue_head_t *vbl_queues;

795  /* number of VBLANK interrupts */
796  /* (driver must alloc the right number of counters) */
797  atomic_t *vblank_count;
798  /* signal list to send on VBLANK */
799  struct drm_vbl_sig_list *vbl_sigs;

801  /* number of signals pending on all crtcs */
802  atomic_t vbl_signal_pending;
803  /* number of users of vblank interrupts per crtc */
804  atomic_t *vblank_refcount;
805  /* protected by dev->vbl_lock, used for wraparound handling */
806  u32 *last_vblank;
807  /* so we don't call enable more than */
808  atomic_t *vblank_enabled;
809  /* Display driver is setting mode */
810  int *vblank_inmodeset;
811  /* Don't wait while crtc is likely disabled */
812  int *vblank_suspend;
813  /* size of vblank counter register */
814  u32 max_vblank_count;
815  int num_crtcs;
816  kmutex_t tasklet_lock;
817  void (*locked_tasklet_func)(struct drm_device *dev);

819  pid_t buf_pgid;
820  drm_agp_head_t *agp;
821  drm_sg_mem_t *sg; /* Scatter gather memory */
822  uint32_t *ctx_bitmap;
823  void *dev_private;
824  unsigned int agp_buffer_token;
825  drm_local_map_t *agp_buffer_map;

827  kstat_t *asoft_ksp; /* kstat support */

829  /* name Drawable information */
830  kmutex_t drw_lock;
831  unsigned int drw_bitfield_length;
832  u32 *drw_bitfield;
833  unsigned int drw_info_length;
834  drm_drawable_info_t **drw_info;

836  /* \name GEM information */
837  /* @{ */
838  kmutex_t object_name_lock;
839  struct idr_list object_name_idr;
840  atomic_t object_count;
841  atomic_t object_memory;
842  atomic_t pin_count;
843  atomic_t pin_memory;
844  atomic_t gtt_count;
845  atomic_t gtt_memory;
846  uint32_t gtt_total;
847  uint32_t invalidate_domains; /* domains pending invalidation */
848  uint32_t flush_domains; /* domains pending flush */
849  /* @} */

851  /*
852  * Saving S3 context
853  */

```

```

854  void *s3_private;
855  };

857  /* Memory management support (drm_memory.c) */
858  void drm_mem_init(void);
859  void drm_mem_uninit(void);
860  void *drm_alloc(size_t, int);
861  void *drm_calloc(size_t, size_t, int);
862  void *drm_realloc(void *, size_t, size_t, int);
863  void drm_free(void *, size_t, int);
864  int drm_ioremap(drm_device_t *, drm_local_map_t *);
865  void drm_ioremapfree(drm_local_map_t *);

867  void drm_core_ioremap(struct drm_local_map *, struct drm_device *);
868  void drm_core_ioremapfree(struct drm_local_map *, struct drm_device *);

870  void drm_pci_free(drm_device_t *, drm_dma_handle_t *);
871  void *drm_pci_alloc(drm_device_t *, size_t, size_t, dma_addr_t, int);

873  struct drm_local_map *drm_core_findmap(struct drm_device *, unsigned long);

875  int drm_context_switch(drm_device_t *, int, int);
876  int drm_context_switch_complete(drm_device_t *, int);
877  int drm_ctxbitmap_init(drm_device_t *);
878  void drm_ctxbitmap_cleanup(drm_device_t *);
879  void drm_ctxbitmap_free(drm_device_t *, int);
880  int drm_ctxbitmap_next(drm_device_t *);

882  /* Locking IOCTL support (drm_lock.c) */
883  int drm_lock_take(drm_lock_data_t *, unsigned int);
884  int drm_lock_transfer(drm_device_t *,
885                      drm_lock_data_t *, unsigned int);
886  int drm_lock_free(drm_device_t *,
887                   volatile unsigned int *, unsigned int);

889  /* Buffer management support (drm_bufs.c) */
890  unsigned long drm_get_resource_start(drm_device_t *, unsigned int);
891  unsigned long drm_get_resource_len(drm_device_t *, unsigned int);
892  int drm_initmap(drm_device_t *, unsigned long, unsigned long,
893                unsigned int, int, int);
894  void drm_rmmmap(drm_device_t *, drm_local_map_t *);
895  int drm_addmap(drm_device_t *, unsigned long, unsigned long,
896               drm_map_type_t, drm_map_flags_t, drm_local_map_t **);
897  int drm_order(unsigned long);

899  /* DMA support (drm_dma.c) */
900  int drm_dma_setup(drm_device_t *);
901  void drm_dma_takedown(drm_device_t *);
902  void drm_free_buffer(drm_device_t *, drm_buf_t *);
903  void drm_reclaim_buffers(drm_device_t *, drm_file_t *);
904  #define drm_core_reclaim_buffers    drm_reclaim_buffers

906  /* IRQ support (drm_irq.c) */
907  int drm_irq_install(drm_device_t *);
908  int drm_irq_uninstall(drm_device_t *);
909  uint_t drm_irq_handler(DRM_IRQ_ARGS);
910  void drm_driver_irq_preinstall(drm_device_t *);
911  void drm_driver_irq_postinstall(drm_device_t *);
912  void drm_driver_irq_uninstall(drm_device_t *);
913  int drm_vblank_wait(drm_device_t *, unsigned int *);
914  void drm_vbl_send_signals(drm_device_t *);
915  void drm_handle_vblank(struct drm_device *dev, int crtc);
916  u32 drm_vblank_count(struct drm_device *dev, int crtc);
917  int drm_vblank_get(struct drm_device *dev, int crtc);
918  void drm_vblank_put(struct drm_device *dev, int crtc);
919  int drm_vblank_init(struct drm_device *dev, int num_crtcs);

```

```

920 void    drm_vblank_cleanup(struct drm_device *dev);
921 int     drm_modeset_ctl(DRM_IOCTL_ARGS);

923 /* AGP/GART support (drm_agpsupport.c) */
924 int     drm_device_is_agp(drm_device_t *);
925 int     drm_device_is_pcie(drm_device_t *);
926 drm_agp_head_t *drm_agp_init(drm_device_t *);
927 void    drm_agp_fini(drm_device_t *);
928 int     drm_agp_do_release(drm_device_t *);
929 void    *drm_agp_allocate_memory(size_t pages,
930    uint32_t type, drm_device_t *dev);
931 int     drm_agp_free_memory(agp_allocate_t *handle, drm_device_t *dev);
932 int     drm_agp_bind_memory(unsigned int, uint32_t, drm_device_t *);
933 int     drm_agp_unbind_memory(unsigned long, drm_device_t *);
934 int     drm_agp_bind_pages(drm_device_t *dev,
935    pfn_t *pages,
936    unsigned long num_pages,
937    uint32_t gtt_offset);
938 int     drm_agp_unbind_pages(drm_device_t *dev,
939    unsigned long num_pages,
940    uint32_t gtt_offset,
941    uint32_t type);
942 void    drm_agp_chipset_flush(struct drm_device *dev);
943 void    drm_agp_rebind(struct drm_device *dev);

945 /* kstat support (drm_kstats.c) */
946 int     drm_init_kstats(drm_device_t *);
947 void    drm_fini_kstats(drm_device_t *);

949 /* Scatter Gather Support (drm_scatter.c) */
950 void    drm_sg_cleanup(drm_device_t *, drm_sg_mem_t *);

952 /* ATI PCIGART support (ati_pcigart.c) */
953 int     drm_ati_pcigart_init(drm_device_t *, drm_ati_pcigart_info *);
954 int     drm_ati_pcigart_cleanup(drm_device_t *, drm_ati_pcigart_info *);

956 /* Locking IOCTL support (drm_drv.c) */
957 int     drm_lock(DRM_IOCTL_ARGS);
958 int     drm_unlock(DRM_IOCTL_ARGS);
959 int     drm_version(DRM_IOCTL_ARGS);
960 int     drm_setversion(DRM_IOCTL_ARGS);
961 /* Cache management (drm_cache.c) */
962 void    drm_clflush_pages(caddr_t *pages, unsigned long num_pages);

964 /* Misc. IOCTL support (drm_ioctl.c) */
965 int     drm_irq_by_busid(DRM_IOCTL_ARGS);
966 int     drm_getunique(DRM_IOCTL_ARGS);
967 int     drm_setunique(DRM_IOCTL_ARGS);
968 int     drm_getmap(DRM_IOCTL_ARGS);
969 int     drm_getclient(DRM_IOCTL_ARGS);
970 int     drm_getstats(DRM_IOCTL_ARGS);
971 int     drm_noop(DRM_IOCTL_ARGS);

973 /* Context IOCTL support (drm_context.c) */
974 int     drm_resctx(DRM_IOCTL_ARGS);
975 int     drm_addctx(DRM_IOCTL_ARGS);
976 int     drm_modctx(DRM_IOCTL_ARGS);
977 int     drm_getctx(DRM_IOCTL_ARGS);
978 int     drm_switchctx(DRM_IOCTL_ARGS);
979 int     drm_newctx(DRM_IOCTL_ARGS);
980 int     drm_rmctx(DRM_IOCTL_ARGS);
981 int     drm_setsareactx(DRM_IOCTL_ARGS);
982 int     drm_getsareactx(DRM_IOCTL_ARGS);

984 /* Drawable IOCTL support (drm_drawable.c) */
985 int     drm_adddraw(DRM_IOCTL_ARGS);

```

```

986 int     drm_rmdraw(DRM_IOCTL_ARGS);
987 int     drm_update_draw(DRM_IOCTL_ARGS);

989 /* Authentication IOCTL support (drm_auth.c) */
990 int     drm_getmagic(DRM_IOCTL_ARGS);
991 int     drm_authmagic(DRM_IOCTL_ARGS);
992 int     drm_remove_magic(drm_device_t *, drm_magic_t);
993 drm_file_t *drm_find_file(drm_device_t *, drm_magic_t);
994 /* Buffer management support (drm_bufs.c) */
995 int     drm_addmap_ioctl(DRM_IOCTL_ARGS);
996 int     drm_rmap_ioctl(DRM_IOCTL_ARGS);
997 int     drm_addbufs_ioctl(DRM_IOCTL_ARGS);
998 int     drm_infobufs(DRM_IOCTL_ARGS);
999 int     drm_markbufs(DRM_IOCTL_ARGS);
1000 int     drm_freebufs(DRM_IOCTL_ARGS);
1001 int     drm_mapbufs(DRM_IOCTL_ARGS);

1003 /* DMA support (drm_dma.c) */
1004 int     drm_dma(DRM_IOCTL_ARGS);

1006 /* IRQ support (drm_irq.c) */
1007 int     drm_control(DRM_IOCTL_ARGS);
1008 int     drm_wait_vblank(DRM_IOCTL_ARGS);

1010 /* AGP/GART support (drm_agpsupport.c) */
1011 int     drm_agp_acquire(DRM_IOCTL_ARGS);
1012 int     drm_agp_release(DRM_IOCTL_ARGS);
1013 int     drm_agp_enable(DRM_IOCTL_ARGS);
1014 int     drm_agp_info(DRM_IOCTL_ARGS);
1015 int     drm_agp_alloc(DRM_IOCTL_ARGS);
1016 int     drm_agp_free(DRM_IOCTL_ARGS);
1017 int     drm_agp_unbind(DRM_IOCTL_ARGS);
1018 int     drm_agp_bind(DRM_IOCTL_ARGS);

1020 /* Scatter Gather Support (drm_scatter.c) */
1021 int     drm_sg_alloc(DRM_IOCTL_ARGS);
1022 int     drm_sg_free(DRM_IOCTL_ARGS);

1024 /*      drm_mm.c      */
1025 struct drm_mm_node *drm_mm_get_block(struct drm_mm_node *parent,
1026    unsigned long size, unsigned alignment);
1027 struct drm_mm_node *drm_mm_search_free(const struct drm_mm *mm,
1028    unsigned long size,
1029    unsigned alignment, int best_match);

1031 extern void drm_mm_clean_ml(const struct drm_mm *mm);
1032 extern int drm_debug_flag;

1034 /* We add function to support DRM_DEBUG, DRM_ERROR, DRM_INFO */
1035 extern void drm_debug(const char *fmt, ...);
1036 extern void drm_error(const char *fmt, ...);
1037 extern void drm_info(const char *fmt, ...);

1039 #ifdef DEBUG
1040 #define DRM_DEBUG          if (drm_debug_flag >= 2) drm_debug
1041 #define DRM_INFO          if (drm_debug_flag >= 1) drm_info
1042 #else
1043 #define DRM_DEBUG(...)
1044 #define DRM_INFO(...)
1045 #endif

1047 #define DRM_ERROR          drm_error

1050 #define MAX_INSTNUMS 16

```

```
1052 extern int drm_dev_to_instance(dev_t);
1053 extern int drm_dev_to_minor(dev_t);
1054 extern void *drm_supp_register(dev_info_t *, drm_device_t *);
1055 extern int drm_supp_unregister(void *);

1057 extern int drm_open(drm_device_t *, drm_cminor_t *, int, int, cred_t *);
1058 extern int drm_close(drm_device_t *, int, int, int, cred_t *);
1059 extern int drm_attach(drm_device_t *);
1060 extern int drm_detach(drm_device_t *);
1061 extern int drm_probe(drm_device_t *, drm_pci_id_list_t *);

1063 extern int drm_pci_init(drm_device_t *);
1064 extern void drm_pci_end(drm_device_t *);
1065 extern int pci_get_info(drm_device_t *, int *, int *, int *);
1066 extern int pci_get_irq(drm_device_t *);
1067 extern int pci_get_vendor(drm_device_t *);
1068 extern int pci_get_device(drm_device_t *);

1070 extern struct drm_drawable_info *drm_get_drawable_info(drm_device_t *,
1071                                                       drm_drawable_t);
1072 /* File Operations helpers (drm_fops.c) */
1073 extern drm_file_t *drm_find_file_by_proc(drm_device_t *, cred_t *);
1074 extern drm_cminor_t *drm_find_file_by_minor(drm_device_t *, int);
1075 extern int drm_open_helper(drm_device_t *, drm_cminor_t *, int, int,
1076                            cred_t *);

1078 /* Graphics Execution Manager library functions (drm_gem.c) */
1079 int drm_gem_init(struct drm_device *dev);
1080 void drm_gem_object_free(struct drm_gem_object *obj);
1081 struct drm_gem_object *drm_gem_object_alloc(struct drm_device *dev,
1082                                             size_t size);
1083 void drm_gem_object_handle_free(struct drm_gem_object *obj);

1085 void drm_gem_object_reference(struct drm_gem_object *obj);
1086 void drm_gem_object_unreference(struct drm_gem_object *obj);

1088 int drm_gem_handle_create(struct drm_file *file_priv,
1089                          struct drm_gem_object *obj,
1090                          int *handlep);
1091 void drm_gem_object_handle_reference(struct drm_gem_object *obj);

1093 void drm_gem_object_handle_unreference(struct drm_gem_object *obj);

1095 struct drm_gem_object *drm_gem_object_lookup(struct drm_file *filp,
1096                                             int handle);
1097 int drm_gem_close_ioctl(DRM_IOCTL_ARGS);
1098 int drm_gem_flink_ioctl(DRM_IOCTL_ARGS);
1099 int drm_gem_open_ioctl(DRM_IOCTL_ARGS);
1100 void drm_gem_open(struct drm_file *file_private);
1101 void drm_gem_release(struct drm_device *dev, struct drm_file *file_private);

1104 #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
*****
1 /*
2  * sfe_util.c: general ethernet mac driver framework version 2.6
3  *
4  * Copyright (c) 2002-2008 Masayuki Murayama. All rights reserved.
5  *
6  * Redistribution and use in source and binary forms, with or without
7  * modification, are permitted provided that the following conditions are met:
8  *
9  * 1. Redistributions of source code must retain the above copyright notice,
10 * this list of conditions and the following disclaimer.
11 *
12 * 2. Redistributions in binary form must reproduce the above copyright notice,
13 * this list of conditions and the following disclaimer in the documentation
14 * and/or other materials provided with the distribution.
15 *
16 * 3. Neither the name of the author nor the names of its contributors may be
17 * used to endorse or promote products derived from this software without
18 * specific prior written permission.
19 *
20 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
21 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
22 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
23 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
24 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
25 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
26 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
27 * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
28 * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
29 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
30 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
31 * DAMAGE.
32 */
33
34 /*
35  * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
36  * Use is subject to license terms.
37  */
38
39 /*
40  * System Header files.
41  */
42 #include <sys/types.h>
43 #include <sys/conf.h>
44 #include <sys/debug.h>
45 #include <sys/kmem.h>
46 #include <sys/vtrace.h>
47 #include <sys/ethernet.h>
48 #include <sys/modctl.h>
49 #include <sys/errno.h>
50 #include <sys/ddi.h>
51 #include <sys/sunddi.h>
52 #include <sys/stream.h> /* required for MBLK* */
53 #include <sys/strsun.h> /* required for mionack() */
54 #include <sys/byteorder.h>
55 #include <sys/sysmacros.h>
56 #endif /* ! codereview */
57 #include <sys/pci.h>
58 #include <inet/common.h>
59 #include <inet/led.h>
60 #include <inet/mi.h>
61 #include <inet/nd.h>

```

```

62 #include <sys/crc32.h>
63
64 #include <sys/note.h>
65
66 #include "sfe_mii.h"
67 #include "sfe_util.h"
68
69
70
71 extern char ident[];
72
73 /* Debugging support */
74 #ifdef GEM_DEBUG_LEVEL
75 static int gem_debug = GEM_DEBUG_LEVEL;
76 #define DPRINTF(n, args) if (gem_debug > (n)) cmn_err args
77 #else
78 #define DPRINTF(n, args)
79 #undef ASSERT
80 #define ASSERT(x)
81 #endif
82
83 #define IOC_LINESIZE 0x40 /* Is it right for amd64? */
84
85 /*
86  * Useful macros and typedefs
87  */
88 #define ROUNDUP(x, a) (((x) + (a) - 1) & ~((a) - 1))
89
90 #define GET_NET16(p) (((uint8_t *) (p))[0] << 8 | ((uint8_t *) (p))[1])
91 #define GET_ETHERTYPE(p) GET_NET16(((uint8_t *) (p)) + ETHERADDRL*2)
92
93 #define GET_IPTYPEv4(p) (((uint8_t *) (p))[sizeof (struct ether_header) + 9])
94 #define GET_IPTYPEv6(p) (((uint8_t *) (p))[sizeof (struct ether_header) + 6])
95
96
97 #ifndef INT32_MAX
98 #define INT32_MAX 0x7fffffff
99 #endif
100
101 #define VTAG_OFF (ETHERADDRL*2)
102 #ifndef VTAG_SIZE
103 #define VTAG_SIZE 4
104 #endif
105 #ifndef VTAG_TPID
106 #define VTAG_TPID 0x8100U
107 #endif
108
109 #define GET_TXBUF(dp, sn) \
110 &(dp)->tx_buf[SLOT((dp)->tx_slots_base + (sn), (dp)->gc_tx_buf_size)]
111
112 #ifndef offsetof
113 #define offsetof(t, m) ((long)&(((t *) 0)->m))
114 #endif
115 #define TXFLAG_VTAG(flag) \
116 (((flag) & GEM_TXFLAG_VTAG) >> GEM_TXFLAG_VTAG_SHIFT)
117
118 #define MAXPKTBUF(dp) \
119 ((dp)->mtu + sizeof (struct ether_header) + VTAG_SIZE + ETHERFCSL)
120
121 #define WATCH_INTERVAL_FAST drv_usecstohz(100*1000) /* 100ms */
122 #define BOOLEAN(x) ((x) != 0)
123
124 /*
125  * Macros to distinct chip generation.
126  */

```

```
125 /*
126  * Private functions
127  */
128 static void gem_mii_start(struct gem_dev *);
129 static void gem_mii_stop(struct gem_dev *);

131 /* local buffer management */
132 static void gem_nd_setup(struct gem_dev *dp);
133 static void gem_nd_cleanup(struct gem_dev *dp);
134 static int gem_alloc_memory(struct gem_dev *);
135 static void gem_free_memory(struct gem_dev *);
136 static void gem_init_rx_ring(struct gem_dev *);
137 static void gem_init_tx_ring(struct gem_dev *);
138 __INLINE__ static void gem_append_rxbuf(struct gem_dev *, struct rxbuf *);

140 static void gem_tx_timeout(struct gem_dev *);
141 static void gem_mii_link_watcher(struct gem_dev *dp);
142 static int gem_mac_init(struct gem_dev *dp);
143 static int gem_mac_start(struct gem_dev *dp);
144 static int gem_mac_stop(struct gem_dev *dp, uint_t flags);
145 static void gem_mac_ioctl(struct gem_dev *dp, queue_t *wq, mblk_t *mp);

147 static struct ether_addr gem_etherbroadcastaddr = {
148     0xff, 0xff, 0xff, 0xff, 0xff, 0xff
149 };
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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25 */

27 #include <sys/stat.h>
28 #include <sys/ddi.h>
29 #include <sys/sunddi.h>
30 #include <sys/time.h>
31 #include <sys/varargs.h>
32 #include <sys/conf.h>
33 #include <sys/modctl.h>
34 #include <sys/cmn_err.h>
35 #include <sys/vnode.h>
36 #include <fs/fs_subr.h>
37 #include <sys/types.h>
38 #include <sys/file.h>
39 #include <sys/disp.h>
40 #include <sys/sdt.h>
41 #include <sys/cred.h>
42 #include <sys/list.h>
43 #include <sys/vscan.h>
44 #include <sys/sysmacros.h>
45 #endif /* ! codereview */

47 #define VS_REQ_MAGIC          0x52515354 /* 'RQST' */

49 #define VS_REQS_DEFAULT      20000 /* pending scan requests - req1 */
50 #define VS_NODES_DEFAULT    128 /* concurrent file scans */
51 #define VS_WORKERS_DEFAULT   32 /* worker threads */
52 #define VS_SCANWAIT_DEFAULT  15*60 /* seconds to wait for scan result */
53 #define VS_REQ_HANDLER_TIMEOUT 30
54 #define VS_EXT_RECURSE_DEPTH 8

56 /* access derived from scan result (VS_STATUS_XXX) and file attributes */
57 #define VS_ACCESS_UNDEFINED  0
58 #define VS_ACCESS_ALLOW     1 /* return 0 */
59 #define VS_ACCESS_DENY     2 /* return EACCES */

61 #define tolower(C)          (((C) >= 'A' && (C) <= 'Z') ? (C) - 'A' + 'a' : (C))

```

```

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

63 /* global variables - tunable via /etc/system */
64 uint32_t vs_reqs_max = VS_REQS_DEFAULT; /* max scan requests */
65 uint32_t vs_nodes_max = VS_NODES_DEFAULT; /* max in-progress scan requests */
66 uint32_t vs_workers = VS_WORKERS_DEFAULT; /* max workers send reqs to vscand */
67 uint32_t vs_scan_wait = VS_SCANWAIT_DEFAULT; /* secs to wait for scan result */

70 /*
71  * vscan_svc_state
72  *
73  * +-----+
74  * | VS_SVC_UNCONFIG |
75  * +-----+
76  *   |               ^
77  *   | svc_init     | svc_fini
78  *   v             |
79  * +-----+
80  * | VS_SVC_IDLE   | <----|
81  * +-----+
82  *   |               |
83  *   | svc_enable   |
84  *   | <-----+
85  *   v             |
86  * +-----+
87  * | VS_SVC_ENABLED | --
88  * +-----+
89  *   |               |
90  *   | svc_disable  | handler thread exit,
91  *   v             | all requests complete
92  * +-----+
93  * | VS_SVC_DISABLED | <----|
94  * +-----+
95  *
96  * svc_enable may occur when we are already in the ENABLED
97  * state if vscand has exited without clean shutdown and
98  * then reconnected within the delayed disable time period
99  * (vs_reconnect_timeout) - see vscan_drv
100 */

102 typedef enum {
103     VS_SVC_UNCONFIG,
104     VS_SVC_IDLE,
105     VS_SVC_ENABLED, /* service enabled and registered */
106     VS_SVC_DISABLED /* service disabled and nunregistered */
107 } 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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License, Version 1.0 only
6  * (the "License"). You may not use this file except in compliance
7  * with the License.
8  *
9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 * or http://www.opensolaris.org/os/licensing.
11 * See the License for the specific language governing permissions
12 * and limitations under the License.
13 *
14 * When distributing Covered Code, include this CDDL HEADER in each
15 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 * If applicable, add the following below this CDDL HEADER, with the
17 * fields enclosed by brackets "[]" replaced with your own identifying
18 * information: Portions Copyright [yyyy] [name of copyright owner]
19 *
20 * CDDL HEADER END
21 */
22 /*
23 * Copyright 2004 Sun Microsystems, Inc. All rights reserved.
24 * Use is subject to license terms.
25 */

27 #ifndef _SYS_ECPPVAR_H
28 #define _SYS_ECPPVAR_H

30 #include <sys/note.h>
31 #include <sys/sysmacros.h>
32 #endif /* ! codereview */

34 #ifdef __cplusplus
35 extern "C" {
36 #endif

38 struct ecppunit;

40 /*
41  * Hardware-abstraction structure
42  */
43 struct ecpp_hw {
44     int      (*map_regs)(struct ecppunit *);      /* map registers */
45     void     (*unmap_regs)(struct ecppunit *);   /* unmap registers */
46     int      (*config_chip)(struct ecppunit *); /* configure SuperIO */
47     void     (*dma_new_mode)(struct ecppunit *); /* config new mode */
48     void     (*mask_intr)(struct ecppunit *);   /* mask interrupts */
49     void     (*unmask_intr)(struct ecppunit *); /* unmask interrupts */
50     int      (*dma_start)(struct ecppunit *);   /* start DMA transfer */
51     int      (*dma_stop)(struct ecppunit *, size_t *); /* stop DMA xfer */
52     size_t   (*dma_getcnt)(struct ecppunit *);  /* get DMA counter */
53     ddi_dma_attr_t *attr;                       /* DMA attributes */
54 };

56 #define ECPP_MAP_REGS(pp)      (pp)->hw->map_regs(pp)
57 #define ECPP_UNMAP_REGS(pp)   (pp)->hw->unmap_regs(pp)
58 #define ECPP_CONFIG_CHIP(pp) (pp)->hw->config_chip(pp)
59 #define ECPP_CONFIG_MODE(pp) (pp)->hw->config_mode(pp)
60 #define ECPP_MASK_INTR(pp)   (pp)->hw->mask_intr(pp)
61 #define ECPP_UNMASK_INTR(pp) (pp)->hw->unmask_intr(pp)

```

```

62 #define ECPP_DMA_START(pp)      (pp)->hw->dma_start(pp)
63 #define ECPP_DMA_STOP(pp, cnt) (pp)->hw->dma_stop(pp, cnt)
64 #define ECPP_DMA_GETCNT(pp)    (pp)->hw->dma_getcnt(pp)

66 /* NSC 87332/97317 and EBus DMAC */
67 struct ecpp_ebus {
68     struct config_reg    *c_reg;      /* configuration registers */
69     ddi_acc_handle_t     c_handle;    /* handle for conf regs */
70     struct cheerio_dma_reg *dmac;    /* ebus dmac registers */
71     ddi_acc_handle_t     d_handle;    /* handle for dmac registers */
72     struct config2_reg   *c2_reg;    /* 97317 2nd level conf regs */
73     ddi_acc_handle_t     c2_handle;   /* handle for c2_reg */
74 };

76 /* Southbridge SuperIO and 8237 DMAC */
77 struct ecpp_ml553 {
78     struct isa_space     *isa_space;  /* all of isa space */
79     ddi_acc_handle_t     d_handle;    /* handle for isa space */
80     uint8_t              chn;        /* 8237 dma channel */
81     int                  isadma_entered; /* Southbridge DMA workaround */
82 };

84 #if defined(__x86)
85 struct ecpp_x86 {
86     uint8_t              chn;
87 };
88 #endif

90 /*
91  * Hardware binding structure
92  */
93 struct ecpp_hw_bind {
94     char      *name;      /* binding name */
95     struct ecpp_hw *hw;   /* hw description */
96     char      *info;     /* info string */
97 };

99 /* ecpp e_busy states */
100 typedef enum {
101     ECPP_IDLE = 1, /* No ongoing transfers */
102     ECPP_BUSY = 2, /* Ongoing transfers on the cable */
103     ECPP_DATA = 3, /* Not used */
104     ECPP_ERR = 4, /* Bad status in Centronics mode */
105     ECPP_FLUSH = 5 /* Currently flushing the q */
106 } ecpp_busy_t;

108 /*
109  * ecpp soft state structure
110  */
111 struct ecppunit {
112     kmutex_t    umutex;      /* lock for this structure */
113     int         instance;    /* instance number */
114     dev_info_t  *dip;        /* device information */
115     ddi_iblock_cookie_t ecpp_trap_cookie; /* interrupt cookie */
116     ecpp_busy_t  e_busy;    /* ecpp busy flag */
117     kcondvar_t  pport_cv;   /* cv to signal idle state */
118     /*
119      * common SuperIO registers
120      */
121     struct info_reg *i_reg;  /* info registers */
122     struct fifo_reg *f_reg;  /* fifo register */
123     ddi_acc_handle_t i_handle;
124     ddi_acc_handle_t f_handle;
125     /*
126      * DMA support
127      */

```

```

128     ddi_dma_handle_t    dma_handle;    /* DMA handle */
129     ddi_dma_cookie_t    dma_cookie;    /* current cookie */
130     uint_t              dma_cookie_count; /* # of cookies */
131     uint_t              dma_nwin;      /* # of DMA windows */
132     uint_t              dma_curwin;    /* current window number */
133     uint_t              dma_dir;      /* transfer direction */
134     /*
135     * hardware-dependent stuff
136     */
137     struct ecpp_hw      *hw;           /* operations/attributes */
138     union {
139         struct ecpp_ebus ebus;        /* hw-dependent data */
140         struct ecpp_ml553 ml553;
141     } #if defined(__x86)
142         struct ecpp_x86 x86;
143 #endif
144     } uh;
145     /*
146     * DDI/STREAMS stuff
147     */
148     boolean_t          oflag;         /* instance open flag */
149     queue_t            *readq;        /* pointer to readq */
150     queue_t            *writeq;       /* pointer to writeq */
151     mblk_t             *msg;          /* current message block */
152     boolean_t          suspended;     /* driver suspended status */
153     /*
154     * Modes of operation
155     */
156     int                current_mode;  /* 1284 mode */
157     uchar_t            current_phase; /* 1284 phase */
158     uchar_t            backchannel;   /* backchannel mode supported */
159     uchar_t            io_mode;       /* transfer mode: PIO/DMA */
160     /*
161     * Ioctls support
162     */
163     struct ecpp_transfer_parms xfer_parms; /* transfer parameters */
164     struct ecpp_regs regs;            /* control/status registers */
165     uint8_t            saved_dsr;     /* store the dsr returned from TESTIO */
166     boolean_t          timeout_error; /* store the timeout for GETERR */
167     uchar_t            port;          /* xfer type: dma/pio/tfifo */
168     struct prn_timeouts prn_timeouts; /* prnio timeouts */
169     /*
170     * ecpp.conf parameters
171     */
172     uchar_t            init_seq;      /* centronics init seq */
173     uint32_t           wsrv_retry;     /* delay (ms) before next wsrv */
174     uint32_t           wait_for_busy;  /* wait for BUSY to deassert */
175     uint32_t           data_setup_time; /* pio centronics handshake */
176     uint32_t           strobe_pulse_width; /* pio centronics handshake */
177     uint8_t            fast_centronics; /* DMA/PIO centronics */
178     uint8_t            fast_compat;    /* DMA/PIO 1284 compatible mode */
179     uint32_t           ecp_rev_speed;  /* rev xfer speed in ECP, bytes/sec */
180     uint32_t           rev_watchdog;   /* rev xfer watchdog period, ms */
181     /*
182     * Timeouts
183     */
184     timeout_id_t       timeout_id;     /* io transfers timer */
185     timeout_id_t       fifo_timer_id;  /* drain SuperIO FIFO */
186     timeout_id_t       wsrv_timer_id;  /* wsrv timeout */
187     /*
188     * Softintr data
189     */
190     ddi_softintr_t     softintr_id;
191     int                softintr_flags; /* flags indicating softintr task */
192     uint8_t            softintr_pending;
193     /*

```

```

194     * Misc stuff
195     */
196     caddr_t            ioblock;       /* transfer buffer block */
197     size_t             xfercnt;       /* # of bytes to transfer */
198     size_t             resid;         /* # of bytes not transferred */
199     caddr_t            next_byte;     /* next byte for PIO transfer */
200     caddr_t            last_byte;     /* last byte for PIO transfer */
201     uint32_t           ecpp_drain_counter; /* allows fifo to drain */
202     dma_cancelled;     /* flushed while dma'ing */
203     uint8_t            tfifo_intr;    /* TFIFO switch interrupt workaround */
204     size_t             nread;         /* requested read */
205     size_t             last_dmacnt;   /* DMA counter value for rev watchdog */
206     uint32_t           rev_timeout_cnt; /* number of watchdog invocations */
207     /*
208     * Spurious interrupt detection
209     */
210     hrtime_t           lastspur;      /* last time spurious intrs started */
211     long               nspur;        /* spurious intrs counter */
212     /*
213     * Statistics
214     */
215     kstat_t            *ksp;          /* kstat pointer */
216     kstat_t            *intrstats;    /* kstat interrupt counter */
217     /*
218     * number of bytes, transferred in and out in each mode
219     */
220     uint32_t           ctxpio_obytes;
221     uint32_t           obytes[ECPP_EPP_MODE+1];
222     uint32_t           ibytes[ECPP_EPP_MODE+1];
223     /*
224     * other stats
225     */
226     uint32_t           to_mode[ECPP_EPP_MODE+1]; /* # transitions to mode */
227     uint32_t           xfer_tout;     /* # transfer timeouts */
228     uint32_t           ctx_cf;        /* #Periph check failures */
229     uint32_t           joblen;        /* of bytes xfer'd since open */
230     uint32_t           isr_reattempt_high; /* max times isr has looped */
231     /*
232     * interrupt stats
233     */
234     uint_t             intr_hard;
235     uint_t             intr_spurious;
236     uint_t             intr_soft;
237     /*
238     * identify second register set for ecp mode on Sx86
239     */
240     int                noecpregs;
241 };
242
243 _NOTE(MUTEX_PROTECTS_DATA(ecppunit::umutex, ecppunit))
244 _NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::dip))
245 _NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::instance))
246 _NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::i_reg))
247 _NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::f_reg))
248 _NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::i_handle))
249 _NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::f_handle))
250 _NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::ecpp_trap_cookie))
251 _NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::readq))
252 _NOTE(DATA_READABLE_WITHOUT_LOCK(ecppunit::writeq))
253
254 /*
255 * current_phase values
256 */
257 #define ECPP_PHASE_INIT      0x00 /* initialization */
258 #define ECPP_PHASE_NEGO     0x01 /* negotiation */
259 #define ECPP_PHASE_TERM     0x02 /* termination */

```

```

260 #define ECPP_PHASE_PO      0x03    /* power-on */
262 #define ECPP_PHASE_C_FWD_DMA 0x10    /* cntrx/compat fwd dma xfer */
263 #define ECPP_PHASE_C_FWD_PIO 0x11    /* cntrx/compat fwd PIO xfer */
264 #define ECPP_PHASE_C_IDLE  0x12    /* cntrx/compat idle */

266 #define ECPP_PHASE_NIBT_REVDATA 0x20 /* nibble/byte reverse data */
267 #define ECPP_PHASE_NIBT_AVAIL  0x21 /* nibble/byte reverse data available */
268 #define ECPP_PHASE_NIBT_NAVAIL 0x22 /* nibble/byte reverse data not avail */
269 #define ECPP_PHASE_NIBT_REVIDLE 0x22 /* nibble/byte reverse idle */
270 #define ECPP_PHASE_NIBT_REVINTR 0x23 /* nibble/byte reverse interrupt */

272 #define ECPP_PHASE_ECP_SETUP 0x30    /* ecp setup */
273 #define ECPP_PHASE_ECP_FWD_XFER 0x31 /* ecp forward transfer */
274 #define ECPP_PHASE_ECP_FWD_IDLE 0x32 /* ecp forward idle */
275 #define ECPP_PHASE_ECP_FWD_REV  0x33 /* ecp forward to reverse */
276 #define ECPP_PHASE_ECP_REV_XFER 0x34 /* ecp reverse transfer */
277 #define ECPP_PHASE_ECP_REV_IDLE 0x35 /* ecp reverse idle */
278 #define ECPP_PHASE_ECP_REV_FWD  0x36 /* ecp reverse to forward */

280 #define ECPP_PHASE_EPP_INIT_IDLE 0x40 /* epp init phase */
281 #define ECPP_PHASE_EPP_IDLE      0x41 /* epp all-round phase */

283 #define FAILURE_PHASE      0x80
284 #define UNDEFINED_PHASE   0x81

286 /* ecpp return values */
287 #define SUCCESS            1
288 #define FAILURE           2

290 #define TRUE               1
291 #define FALSE              0

293 /* message type */
294 #define ECPP_BACKCHANNEL  0x45

296 /* transfer modes */
297 #define ECPP_DMA           0x1
298 #define ECPP_PIO          0x2

300 /* tuneable timing defaults */
301 #define CENTRONICS_RETRY  750    /* 750 milliseconds */
302 #define WAIT_FOR_BUSY    1000   /* 1000 microseconds */
303 #define SUSPEND_TOUT     10     /* # seconds before suspend fails */

305 /* Centronics handshaking defaults */
306 #define DATA_SETUP_TIME  2     /* 2 uSec Data Setup Time (2x min) */
307 #define STROBE_PULSE_WIDTH 2    /* 2 uSec Strobe Pulse (2x min) */

309 /* 1284 Extensibility Request values */
310 #define ECPP_XREQ_NIBBLE  0x00    /* Nibble Mode Rev Channel Transfer */
311 #define ECPP_XREQ_BYTE    0x01    /* Byte Mode Rev Channel Transfer */
312 #define ECPP_XREQ_ID     0x04    /* Request Device ID */
313 #define ECPP_XREQ_ECP    0x10    /* Request ECP Mode */
314 #define ECPP_XREQ_ECPRLE 0x30    /* Request ECP Mode with RLE */
315 #define ECPP_XREQ_EPP    0x40    /* Request EPP Mode */
316 #define ECPP_XREQ_XLINK  0x80    /* Request Extensibility Link */

318 /* softintr flags */
319 #define ECPP_SOFTINTR_PIONEXT 0x1  /* write next byte in PIO mode */

321 /* Stream defaults */
322 #define IO_BLOCK_SZ      1024 * 128 /* transfer buffer size */
323 #define ECPPHIWAT        32 * 1024 * 6
324 #define ECPPLOWAT        32 * 1024 * 4

```

```

326 /* Loop timers */
327 #define ECPP_REG_WRITE_MAX_LOOP 100 /* cpu is faster than superio */
328 #define ECPP_ISR_MAX_DELAY      30  /* DMAC slow PENDING status */

330 /* misc constants */
331 #define ECPP_FIFO_SZ            16   /* FIFO size */
332 #define ECPP_FIFO_DRAIN_PERIOD 250000 /* max FIFO drain period in usec */
333 #define NIBBLE_REV_BLKSZ      1024 /* send up to # bytes at a time */
334 #define FWD_TIMEOUT_DEFAULT   90   /* forward xfer timeout in seconds */
335 #define REV_TIMEOUT_DEFAULT    0    /* reverse xfer timeout in seconds */

337 /* ECP mode constants */
338 #define ECP_REV_BLKSZ         1024 /* send up to # bytes at a time */
339 #define ECP_REV_BLKSZ_MAX    (4 * 1024) /* maximum of # bytes */
340 #define ECP_REV_SPEED        (1 * 1024 * 1024) /* bytes/sec */
341 #define ECP_REV_MINTOUT      5     /* min ECP rev xfer timeout in ms */
342 #define REV_WATCHDOG         100   /* poll DMA counter every # ms */

344 /* spurious interrupt detection */
345 #define SPUR_CRITICAL        100   /* number of interrupts... */
346 #define SPUR_PERIOD          1000000000 /* in # ns */

348 /*
349 * Copyin/copyout states
350 */
351 #define ECPP_STRUCTIN        0
352 #define ECPP_STRUCTOUT      1
353 #define ECPP_ADDRIN         2
354 #define ECPP_ADDRROUT       3

356 /*
357 * As other ioctls require the same structure, put inner struct's into union
358 */
359 struct ecpp_copystate {
360     int     state; /* see above */
361     void    *uaddr; /* user address of the following structure */
362     union {
363         struct ecpp_device_id      devid;
364         struct prn_l284_device_id  prn_devid;
365         struct prn_interface_info  prn_if;
366     } un;
367 };

369 /*
370 * The structure is dynamically created for each M_IOCTL and is bound to mblk
371 */
372 _NOTE(SCHEME_PROTECTS_DATA("unique per call", ecpp_copystate))

374 /* kstat structure */
375 struct ecppkstat {
376     /*
377      * number of bytes, transferred in and out in each mode
378      */
379     struct kstat_named      ek_ctx_obytes;
380     struct kstat_named      ek_ctxpio_obytes;
381     struct kstat_named      ek_nib_obytes;
382     struct kstat_named      ek_ecp_obytes;
383     struct kstat_named      ek_ecp_obytes;
384     struct kstat_named      ek_epp_obytes;
385     struct kstat_named      ek_epp_obytes;
386     struct kstat_named      ek_diag_obytes;
387     /*
388      * number of transitions to particular mode
389      */
390     struct kstat_named      ek_to_ctx;
391     struct kstat_named      ek_to_nib;

```

```

392 struct kstat_named ek_to_ecp;
393 struct kstat_named ek_to_epp;
394 struct kstat_named ek_to_diag;
395 /*
396  * other stats
397  */
398 struct kstat_named ek_xfer_tout; /* # transfer timeouts */
399 struct kstat_named ek_ctx_cf; /* # periph check failures */
400 struct kstat_named ek_joblen; /* # bytes xfer'd since open */
401 struct kstat_named ek_isr_reattempt_high; /* max # times */
402 /* isr has looped */
403 struct kstat_named ek_mode; /* 1284 mode */
404 struct kstat_named ek_phase; /* 1284 ECP phase */
405 struct kstat_named ek_backchan; /* backchannel mode supported */
406 struct kstat_named ek_iomode; /* transfer mode: pio/dma */
407 struct kstat_named ek_state; /* ecpp busy flag */
408 };

410 /* Macros for superio programming */
411 #define PP_PUTB(x, y, z) ddi_put8(x, y, z)
412 #define PP_GETB(x, y) ddi_get8(x, y)

414 #define DSR_READ(pp) PP_GETB((pp)->i_handle, &(pp)->i_reg->dsr)
415 #define DCR_READ(pp) PP_GETB((pp)->i_handle, &(pp)->i_reg->dcr)
416 #define ECR_READ(pp) \
417 (pp->noecpregs) ? 0xff : PP_GETB((pp)->f_handle, &(pp)->f_reg->ecr)
418 #define DATAR_READ(pp) PP_GETB((pp)->i_handle, &(pp)->i_reg->ir.datar)
419 #define DFIFO_READ(pp) \
420 (pp->noecpregs) ? 0xff : PP_GETB((pp)->f_handle, &(pp)->f_reg->fr.dfifo)
421 #define TFIFO_READ(pp) \
422 (pp->noecpregs) ? 0xff : PP_GETB((pp)->f_handle, &(pp)->f_reg->fr.tfifo)

424 #define DCR_WRITE(pp, val) PP_PUTB((pp)->i_handle, &(pp)->i_reg->dcr, val)
425 #define ECR_WRITE(pp, val) \
426 if (!pp->noecpregs) PP_PUTB((pp)->f_handle, &(pp)->f_reg->ecr, val)
427 #define DATAR_WRITE(pp, val) \
428 PP_PUTB((pp)->i_handle, &(pp)->i_reg->ir.datar, val)
429 #define DFIFO_WRITE(pp, val) \
430 if (!pp->noecpregs) PP_PUTB((pp)->f_handle, &(pp)->f_reg->fr.dfifo, val)
431 #define TFIFO_WRITE(pp, val) \
432 if (!pp->noecpregs) PP_PUTB((pp)->f_handle, &(pp)->f_reg->fr.tfifo, val)

434 /*
435  * Macros to manipulate register bits
436  */
437 #define OR_SET_BYTE_R(handle, addr, val) \
438 { \
439 uint8_t tmpval; \
440 tmpval = ddi_get8(handle, (uint8_t *)addr); \
441 tmpval |= val; \
442 ddi_put8(handle, (uint8_t *)addr, tmpval); \
443 }

445 #define OR_SET_LONG_R(handle, addr, val) \
446 { \
447 uint32_t tmpval; \
448 tmpval = ddi_get32(handle, (uint32_t *)addr); \
449 tmpval |= val; \
450 ddi_put32(handle, (uint32_t *)addr, tmpval); \
451 }

453 #define AND_SET_BYTE_R(handle, addr, val) \
454 { \
455 uint8_t tmpval; \
456 tmpval = ddi_get8(handle, (uint8_t *)addr); \
457 tmpval &= val; \

```

```

458 ddi_put8(handle, (uint8_t *)addr, tmpval); \
459 }

461 #define AND_SET_LONG_R(handle, addr, val) \
462 { \
463 uint32_t tmpval; \
464 tmpval = ddi_get32(handle, (uint32_t *)addr); \
465 tmpval &= val; \
466 ddi_put32(handle, (uint32_t *)addr, tmpval); \
467 }

469 #define NOR_SET_LONG_R(handle, addr, val, mask) \
470 { \
471 uint32_t tmpval; \
472 tmpval = ddi_get32(handle, (uint32_t *)addr); \
473 tmpval &= ~(mask); \
474 tmpval |= val; \
475 ddi_put32(handle, (uint32_t *)addr, tmpval); \
476 }

478 /*
479  * Macros for Cheerio/RIO DMAC programming
480  */
481 #define SET_DMAC_CSR(pp, val) ddi_put32(pp->uh.ebus.d_handle, \
482 ((uint32_t *)&pp->uh.ebus.dmac->csr), \
483 ((uint32_t)val))
484 #define GET_DMAC_CSR(pp) ddi_get32(pp->uh.ebus.d_handle, \
485 (uint32_t *)&(pp->uh.ebus.dmac->csr))

487 #define SET_DMAC_ACR(pp, val) ddi_put32(pp->uh.ebus.d_handle, \
488 ((uint32_t *)&pp->uh.ebus.dmac->acr), \
489 ((uint32_t)val))

491 #define GET_DMAC_ACR(pp) ddi_get32(pp->uh.ebus.d_handle, \
492 (uint32_t *)&pp->uh.ebus.dmac->acr)

494 #define SET_DMAC_BCR(pp, val) ddi_put32(pp->uh.ebus.d_handle, \
495 ((uint32_t *)&pp->uh.ebus.dmac->bcr), \
496 ((uint32_t)val))

498 #define GET_DMAC_BCR(pp) ddi_get32(pp->uh.ebus.d_handle, \
499 ((uint32_t *)&pp->uh.ebus.dmac->bcr))

501 #define DMAC_RESET_TIMEOUT 10000 /* in usec */

503 /*
504  * Macros to distinguish between PIO and DMA Compatibility mode
505  */
506 #define COMPAT_PIO(pp) (((pp)->io_mode == ECPP_PIO) && \
507 ((pp)->current_mode == ECPP_CENTRONICS || \
508 (pp)->current_mode == ECPP_COMPAT_MODE))

510 #define COMPAT_DMA(pp) (((pp)->io_mode == ECPP_DMA) && \
511 ((pp)->current_mode == ECPP_CENTRONICS || \
512 (pp)->current_mode == ECPP_COMPAT_MODE))

514 /*
515  * Other useful macros
516  */
517 #define NELEM(a) (sizeof (a) / sizeof (*(a)))
518 #define offsetof(s, m) ((size_t)((s *)0->m))

519 #ifdef __cplusplus
520 }
521 #endif

```

unchanged_portion_omitted

new/usr/src/uts/common/sys/ib/clients/of/sol_ofs/sol_cma.h

1

```
*****
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
*****
```

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
24 */

26 #ifndef _SYS_IB_CLIENTS_OF_SOL_OFS_SOL_CMA_H
27 #define _SYS_IB_CLIENTS_OF_SOL_OFS_SOL_CMA_H

29 #ifdef __cplusplus
30 extern "C" {
31 #endif

33 #include <sys/sysmacros.h>
34 #endif /* ! codereview */

36 #include <sys/ib/clients/of/sol_ofs/sol_ofs_common.h>
37 #include <sys/ib/clients/of/rdma/rdma_cm.h>
38 #include <sys/ib/clients/of/sol_ofs/sol_ib_cma.h> /* Transport Specific */

33 #if !defined(offsetof)
34 #define offsetof(s, m)      (size_t)&(((s *)0)->m)
35 #endif

41 #define IS_UDP_CMID(idp)      ((idp)->ps == RDMA_PS_UDP || \
42 (idp)->ps == RDMA_PS_IPOIB)
43 #define IS_VALID_SOCKADDR(sockaddr) \
44 ((sockaddr)->sa_family == AF_INET || \
45 (sockaddr)->sa_family == AF_INET6)

47 /*
48  * Global structure which contains information about all
49  * CMIDs, which have called rdma_listen().
50  */
51 typedef struct sol_cma_glbl_listen_s {
52     avl_node_t      cma_listen_node;

54     uint64_t        cma_listen_chan_sid;
55     void             *cma_listen_clnt_hdl;
56     void             *cma_listen_svc_hdl;
57     genlist_t        cma_listen_chan_list;
58 } sol_cma_glbl_listen_t;
unchanged_portion_omitted
```

new/usr/src/uts/common/sys/sysmacros.h

1

```
*****
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>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
22 /*      All Rights Reserved      */

25 /*
26 * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
27 * Use is subject to license terms.
28 *
29 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
30 */

32 #ifndef _SYS_SYSMACROS_H
33 #define _SYS_SYSMACROS_H

35 #include <sys/param.h>

37 #ifdef __cplusplus
38 extern "C" {
39 #endif

41 /*
42  * Some macros for units conversion
43  */
44 /*
45  * Disk blocks (sectors) and bytes.
46  */
47 #define dtob(DD)      ((DD) << DEV_BSHIFT)
48 #define btod(BB)      (((BB) + DEV_BSIZE - 1) >> DEV_BSHIFT)
49 #define btodt(BB)     ((BB) >> DEV_BSHIFT)
50 #define lbtod(BB)     (((offset_t)(BB) + DEV_BSIZE - 1) >> DEV_BSHIFT)

52 /* common macros */
53 #ifndef MIN
54 #define MIN(a, b)      ((a) < (b) ? (a) : (b))
55 #endif
56 #ifndef MAX
57 #define MAX(a, b)      ((a) < (b) ? (b) : (a))
58 #endif
59 #ifndef ABS
60 #define ABS(a)         ((a) < 0 ? -(a) : (a))
```

new/usr/src/uts/common/sys/sysmacros.h

2

```
61 #endif
62 #ifndef SIGNOF
63 #define SIGNOF(a)      ((a) < 0 ? -1 : (a) > 0)
64 #endif

66 #ifdef _KERNEL

68 /*
69  * Convert a single byte to/from binary-coded decimal (BCD).
70  */
71 extern unsigned char byte_to_bcd[256];
72 extern unsigned char bcd_to_byte[256];

74 #define BYTE_TO_BCD(x)  byte_to_bcd[(x) & 0xff]
75 #define BCD_TO_BYTE(x)  bcd_to_byte[(x) & 0xff]

77 #endif /* _KERNEL */

79 /*
80  * WARNING: The device number macros defined here should not be used by device
81  * drivers or user software. Device drivers should use the device functions
82  * defined in the DDI/DKI interface (see also ddi.h). Application software
83  * should make use of the library routines available in makedev(3). A set of
84  * new device macros are provided to operate on the expanded device number
85  * format supported in SVR4. Macro versions of the DDI device functions are
86  * provided for use by kernel proper routines only. Macro routines bmajor(),
87  * major(), minor(), emajor(), eminor(), and makedev() will be removed or
88  * their definitions changed at the next major release following SVR4.
89  */

91 #define O_BITSMAJOR      7      /* # of SVR3 major device bits */
92 #define O_BITSMINOR     8      /* # of SVR3 minor device bits */
93 #define O_MAXMAJ        0x7f   /* SVR3 max major value */
94 #define O_MAXMIN        0xff   /* SVR3 max minor value */

97 #define L_BITSMAJOR32   14     /* # of SVR4 major device bits */
98 #define L_BITSMINOR32  18     /* # of SVR4 minor device bits */
99 #define L_MAXMAJ32     0x3fff  /* SVR4 max major value */
100 #define L_MAXMIN32    0x3ffff  /* MAX minor for 3b2 software drivers. */
101 /* For 3b2 hardware devices the minor is */
102 /* restricted to 256 (0-255) */

104 #ifdef _LP64
105 #define L_BITSMAJOR     32     /* # of major device bits in 64-bit Solaris */
106 #define L_BITSMINOR    32     /* # of minor device bits in 64-bit Solaris */
107 #define L_MAXMAJ       0xfffffffful /* max major value */
108 #define L_MAXMIN       0xfffffffful /* max minor value */
109 #else
110 #define L_BITSMAJOR     L_BITSMAJOR32
111 #define L_BITSMINOR    L_BITSMINOR32
112 #define L_MAXMAJ       L_MAXMAJ32
113 #define L_MAXMIN       L_MAXMIN32
114 #endif

116 #ifdef _KERNEL

118 /* major part of a device internal to the kernel */

120 #define major(x)        (major_t)((((unsigned)(x)) >> O_BITSMINOR) & O_MAXMAJ)
121 #define bmajor(x)       (major_t)((((unsigned)(x)) >> O_BITSMINOR) & O_MAXMAJ)

123 /* get internal major part of expanded device number */

125 #define getmajor(x)     (major_t)((((dev_t)(x)) >> L_BITSMINOR) & L_MAXMAJ)
```

```

127 /* minor part of a device internal to the kernel */
129 #define minor(x)      (minor_t)((x) & O_MAXMIN)
131 /* get internal minor part of expanded device number */
133 #define getminor(x)   (minor_t)((x) & L_MAXMIN)
135 #else /* _KERNEL */
137 /* major part of a device external from the kernel (same as emajor below) */
139 #define major(x)      (major_t)((((unsigned)(x)) >> O_BITSMINOR) & O_MAXMAJ)
141 /* minor part of a device external from the kernel (same as eminor below) */
143 #define minor(x)      (minor_t)((x) & O_MAXMIN)
145 #endif /* _KERNEL */
147 /* create old device number */
149 #define makedev(x, y) (unsigned short)((((x) << O_BITSMINOR) | ((y) & O_MAXMIN)))
151 /* make an new device number */
153 #define makedevice(x, y) (dev_t)((((dev_t)(x) << L_BITSMINOR) | ((y) & L_MAXMIN)))
155 /*
156 * emajor() allows kernel/driver code to print external major numbers
157 * eminor() allows kernel/driver code to print external minor numbers
158 */
161 #define emajor(x) \
162     (major_t)((((unsigned int)(x) >> O_BITSMINOR) > O_MAXMAJ) ? \
163     NODEV : (((unsigned int)(x) >> O_BITSMINOR) & O_MAXMAJ)
165 #define eminor(x) \
166     (minor_t)((x) & O_MAXMIN)
168 /*
169 * get external major and minor device
170 * components from expanded device number
171 */
172 #define getemajor(x) (major_t)((((dev_t)(x) >> L_BITSMINOR) > L_MAXMAJ) ? \
173     NODEV : (((dev_t)(x) >> L_BITSMINOR) & L_MAXMAJ))
174 #define geteminor(x) (minor_t)((x) & L_MAXMIN)
176 /*
177 * These are versions of the kernel routines for compressing and
178 * expanding long device numbers that don't return errors.
179 */
180 #if (L_BITSMAJOR32 == L_BITSMAJOR) && (L_BITSMINOR32 == L_BITSMINOR)
182 #define DEVCML(x)      (x)
183 #define DEVEXPL(x)    (x)
185 #else
187 #define DEVCML(x) \
188     (dev32_t)((((x) >> L_BITSMINOR) > L_MAXMAJ32 || \
189     ((x) & L_MAXMIN) > L_MAXMIN32) ? NODEV32 : \
190     (((x) >> L_BITSMINOR) << L_BITSMINOR32) | ((x) & L_MAXMIN32)))
192 #define DEVEXPL(x) \

```

```

193     (((x) == NODEV32) ? NODEV : \
194     makedevice(((x) >> L_BITSMINOR32) & L_MAXMAJ32, (x) & L_MAXMIN32))
196 #endif /* L_BITSMAJOR32 ... */
198 /* convert to old (SVR3.2) dev format */
200 #define cmpdev(x) \
201     (o_dev_t)((((x) >> L_BITSMINOR) > O_MAXMAJ || \
202     ((x) & L_MAXMIN) > O_MAXMIN) ? NODEV : \
203     (((x) >> L_BITSMINOR) << O_BITSMINOR) | ((x) & O_MAXMIN))
205 /* convert to new (SVR4) dev format */
207 #define expdev(x) \
208     (dev_t)((((dev_t)((x) >> O_BITSMINOR) & O_MAXMAJ) << L_BITSMINOR) | \
209     ((x) & O_MAXMIN))
211 /*
212 * Macro for checking power of 2 address alignment.
213 */
214 #define IS_P2ALIGNED(v, a) (((uintptr_t)(v) & ((uintptr_t)(a) - 1)) == 0)
216 /*
217 * Macros for counting and rounding.
218 */
219 #define howmany(x, y) (((x)+(y)-1)/(y))
220 #define roundup(x, y) (((x)+(y)-1)/(y))*y)
222 /*
223 * Macro to determine if value is a power of 2
224 */
225 #define ISP2(x)      ((x) & ((x) - 1)) == 0)
227 /*
228 * Macros for various sorts of alignment and rounding. The "align" must
229 * be a power of 2. Often times it is a block, sector, or page.
230 */
232 /*
233 * return x rounded down to an align boundary
234 * eg, P2ALIGN(1200, 1024) == 1024 (1*align)
235 * eg, P2ALIGN(1024, 1024) == 1024 (1*align)
236 * eg, P2ALIGN(0x1234, 0x100) == 0x1200 (0x12*align)
237 * eg, P2ALIGN(0x5600, 0x100) == 0x5600 (0x56*align)
238 */
239 #define P2ALIGN(x, align)      ((x) & ~(align))
241 /*
242 * return x % (mod) align
243 * eg, P2PHASE(0x1234, 0x100) == 0x34 (x-0x12*align)
244 * eg, P2PHASE(0x5600, 0x100) == 0x00 (x-0x56*align)
245 */
246 #define P2PHASE(x, align)      ((x) & ((align) - 1))
248 /*
249 * return how much space is left in this block (but if it's perfectly
250 * aligned, return 0).
251 * eg, P2NPHASE(0x1234, 0x100) == 0xcc (0x13*align-x)
252 * eg, P2NPHASE(0x5600, 0x100) == 0x00 (0x56*align-x)
253 */
254 #define P2NPHASE(x, align)      (-(x) & ((align) - 1))
256 /*
257 * return x rounded up to an align boundary
258 * eg, P2ROUNDUP(0x1234, 0x100) == 0x1300 (0x13*align)

```

```

259 * eg, P2ROUNDUP(0x5600, 0x100) == 0x5600 (0x56*align)
260 */
261 #define P2ROUNDUP(x, align)          (-(~(x) & ~(align)))

263 /*
264 * return the ending address of the block that x is in
265 * eg, P2END(0x1234, 0x100) == 0x12ff (0x13*align - 1)
266 * eg, P2END(0x5600, 0x100) == 0x56ff (0x57*align - 1)
267 */
268 #define P2END(x, align)              (-(~(x) & ~(align)))

270 /*
271 * return x rounded up to the next phase (offset) within align.
272 * phase should be < align.
273 * eg, P2PHASEUP(0x1234, 0x100, 0x10) == 0x1310 (0x13*align + phase)
274 * eg, P2PHASEUP(0x5600, 0x100, 0x10) == 0x5610 (0x56*align + phase)
275 */
276 #define P2PHASEUP(x, align, phase)  ((phase) - (((phase) - (x)) & ~(align)))

278 /*
279 * return TRUE if adding len to off would cause it to cross an align
280 * boundary.
281 * eg, P2BOUNDARY(0x1234, 0xe0, 0x100) == TRUE (0x1234 + 0xe0 == 0x1314)
282 * eg, P2BOUNDARY(0x1234, 0x50, 0x100) == FALSE (0x1234 + 0x50 == 0x1284)
283 */
284 #define P2BOUNDARY(off, len, align) \
285     (((off) ^ ((off) + (len) - 1)) > (align) - 1)

287 /*
288 * Return TRUE if they have the same highest bit set.
289 * eg, P2SAMEHIGHBIT(0x1234, 0x1001) == TRUE (the high bit is 0x1000)
290 * eg, P2SAMEHIGHBIT(0x1234, 0x3010) == FALSE (high bit of 0x3010 is 0x2000)
291 */
292 #define P2SAMEHIGHBIT(x, y)         (((x) ^ (y)) < ((x) & (y)))

294 /*
295 * Typed version of the P2* macros. These macros should be used to ensure
296 * that the result is correctly calculated based on the data type of (x),
297 * which is passed in as the last argument, regardless of the data
298 * type of the alignment. For example, if (x) is of type uint64_t,
299 * and we want to round it up to a page boundary using "PAGESIZE" as
300 * the alignment, we can do either
301 *     P2ROUNDUP(x, (uint64_t)PAGESIZE)
302 * or
303 *     P2ROUNDUP_TYPED(x, PAGESIZE, uint64_t)
304 */
305 #define P2ALIGN_TYPED(x, align, type) \
306     ((type)(x) & ~(type)(align))
307 #define P2PHASE_TYPED(x, align, type) \
308     ((type)(x) & ((type)(align) - 1))
309 #define P2NPHASE_TYPED(x, align, type) \
310     (~(type)(x) & ((type)(align) - 1))
311 #define P2ROUNDUP_TYPED(x, align, type) \
312     (-(~(type)(x) & ~(type)(align)))
313 #define P2END_TYPED(x, align, type) \
314     (-(~(type)(x) & ~(type)(align)))
315 #define P2PHASEUP_TYPED(x, align, phase, type) \
316     ((type)(phase) - (((type)(phase) - (type)(x)) & ~(type)(align)))
317 #define P2CROSS_TYPED(x, y, align, type) \
318     (((type)(x) ^ (type)(y)) > (type)(align) - 1)
319 #define P2SAMEHIGHBIT_TYPED(x, y, type) \
320     (((type)(x) ^ (type)(y)) < ((type)(x) & (type)(y)))

322 /*
323 * Macros to atomically increment/decrement a variable. mutex and var
324 * must be pointers.

```

```

325 */
326 #define INCR_COUNT(var, mutex) mutex_enter(mutex), (*(var))++, mutex_exit(mutex)
327 #define DECR_COUNT(var, mutex) mutex_enter(mutex), (*(var))--, mutex_exit(mutex)

329 /*
330 * Macros to declare bitfields - the order in the parameter list is
331 * Low to High - that is, declare bit 0 first. We only support 8-bit bitfields
332 * because if a field crosses a byte boundary it's not likely to be meaningful
333 * without reassembly in its nonnative endianness.
334 */
335 #if defined(_BIT_FIELDS_LTOH)
336 #define DECL_BITFIELD2(_a, _b) \
337     uint8_t _a, _b
338 #define DECL_BITFIELD3(_a, _b, _c) \
339     uint8_t _a, _b, _c
340 #define DECL_BITFIELD4(_a, _b, _c, _d) \
341     uint8_t _a, _b, _c, _d
342 #define DECL_BITFIELD5(_a, _b, _c, _d, _e) \
343     uint8_t _a, _b, _c, _d, _e
344 #define DECL_BITFIELD6(_a, _b, _c, _d, _e, _f) \
345     uint8_t _a, _b, _c, _d, _e, _f
346 #define DECL_BITFIELD7(_a, _b, _c, _d, _e, _f, _g) \
347     uint8_t _a, _b, _c, _d, _e, _f, _g
348 #define DECL_BITFIELD8(_a, _b, _c, _d, _e, _f, _g, _h) \
349     uint8_t _a, _b, _c, _d, _e, _f, _g, _h
350 #elif defined(_BIT_FIELDS_HTOH)
351 #define DECL_BITFIELD2(_a, _b) \
352     uint8_t _b, _a
353 #define DECL_BITFIELD3(_a, _b, _c) \
354     uint8_t _c, _b, _a
355 #define DECL_BITFIELD4(_a, _b, _c, _d) \
356     uint8_t _d, _c, _b, _a
357 #define DECL_BITFIELD5(_a, _b, _c, _d, _e) \
358     uint8_t _e, _d, _c, _b, _a
359 #define DECL_BITFIELD6(_a, _b, _c, _d, _e, _f) \
360     uint8_t _f, _e, _d, _c, _b, _a
361 #define DECL_BITFIELD7(_a, _b, _c, _d, _e, _f, _g) \
362     uint8_t _g, _f, _e, _d, _c, _b, _a
363 #define DECL_BITFIELD8(_a, _b, _c, _d, _e, _f, _g, _h) \
364     uint8_t _h, _g, _f, _e, _d, _c, _b, _a
365 #else
366 #error One of _BIT_FIELDS_LTOH or _BIT_FIELDS_HTOH must be defined
367 #endif /* _BIT_FIELDS_LTOH */

369 /* avoid any possibility of clashing with <stddef.h> version */
370 #if (defined(_KERNEL) || defined(_FAKE_KERNEL)) && !defined(_KMEMUSER)

372 #if !defined(offsetof)
373 #if __GNUC__ > 4 || (__GNUC__ == 4 && __GNUC_MINOR__ >= 5)
374 #define offsetof(s, m) __builtin_offsetof(s, m)
375 #else
376 #endif /* ! codereview */
377 #define offsetof(s, m) (((size_t)((s *)0)->m))
378 #endif
379 #endif /* ! codereview */
380 #endif /* !offsetof */

382 #define container_of(m, s, name) \
383     (void *)((uintptr_t)(m) - (uintptr_t)offsetof(s, name))

385 #define ARRAY_SIZE(x) (sizeof (x) / sizeof (x[0]))
386 #endif /* _KERNEL, !_KMEMUSER */

388 #ifdef __cplusplus
389 }
390 #endif

```



```
392 #endif /* _SYS_SYSMACROS_H */
```

new/usr/src/uts/common/sys/usb/clients/audio/usb_ac/usb_ac.h

1

```
*****
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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 #ifndef _SYS_USB_AC_H
27 #define _SYS_USB_AC_H

31 #ifdef __cplusplus
32 extern "C" {
33 #endif

35 #include <sys/sunldi.h>
36 #include <sys/sysmacros.h>
37 #endif /* !codereview */
38 #include <sys/usb/usba/usbai_private.h>

41 int usb_ac_open(dev_info_t *);
42 void usb_ac_close(dev_info_t *);

45 /* structure for each unit described by descriptors */
46 typedef struct usb_ac_unit_list {
47     uint_t      acu_type;
48     void        *acu_descriptor;
49     size_t      acu_descr_length;
50 } usb_ac_unit_list_t;

52 #define USB_AC_ID_NONE                0

54 #define USB_AC_FIND_ONE                0
55 #define USB_AC_FIND_ALL                1
56 #define USB_AC_MAX_DEPTH                8

58 /*
59 * plumbing data; info per plumbed module
60 */
61 typedef struct usb_ac_plumbed {
```

new/usr/src/uts/common/sys/usb/clients/audio/usb_ac/usb_ac.h

2

```
62     struct usb_ac_state *acp_uacp; /* usb_ac state pointer */
63     dev_info_t      *acp_dip; /* devinfo pointer */
64     uint_t          acp_ifno; /* interface number */
65     int             acp_driver; /* Plumbed driver, see value below */

67     ldi_handle_t    acp_lh; /* ldi handle of plumbed driver */
68     dev_t           acp_devt; /* devt of plumbed driver */
69     ddi_taskq_t     *acp_tqp; /* taskq for I/O to plumbed driver */
70     int             acp_flags;
71 #define ACP_ENABLED 1

73     void            *acp_data; /* ptr to streams or hid data */
74 } usb_ac_plumbed_t;

77 /*
78 * request structure to usb_as: info per MCTL request;
79 * only one active at a time.
80 */
81 typedef struct usb_ac_to_as_req {
82     usb_audio_formats_t acr_curr_format; /* format data from mixer */
83 } usb_ac_to_as_req_t;

86 /* registration and plumbing info per streaming interface */
87 typedef struct usb_ac_streams_info {
88     usb_ac_plumbed_t *acs_plumbed; /* ptr to entry in plumbed list */
89     uint_t            acs_rcvd_reg_data; /* valid registration data rcvd */
90     void              *acs_ptr; /* pointer to registration data */
91     usb_as_registration_t acs_streams_reg;
92 } usb_ac_streams_info_t;

96 /* Multiple command management */
97 int acs_setup_teardown_count;

99 uint8_t acs_default_gain;
100 } usb_ac_streams_info_t;

103 /* power state */
104 typedef struct usb_ac_power {
105     void        *acpm_state; /* points back to usb_ac_state */
106     int         acpm_pm_busy; /* device busy accounting */
107     uint8_t     acpm_wakeup_enabled;

109     /* this is the bit mask of the power states that device has */
110     uint8_t     acpm_pwr_states;

112     /* wakeup and power transition capabilities of an interface */
113     uint8_t     acpm_capabilities;

115     /* current power level the device is in */
116     uint8_t     acpm_current_power;
117 } usb_ac_power_t;

119 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_power_t::acpm_state))
120 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_power_t::acpm_wakeup_enabled))
121 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_power_t::acpm_pwr_states))
122 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_power_t::acpm_capabilities))

124 typedef struct usb_audio_format {
125     int         sr; /* sample rate */
126     uint_t      ch; /* channels */
127     uint_t      prec; /* precision */
```

```

128     uint_t          enc;      /* encoding */
129 } usb_audio_format_t;

132 typedef struct usb_audio_eng {
133     void *statep;
134     usb_ac_streams_info_t *streams;
135     audio_engine_t *af_engp;

137     int             af_eflags; /* ENGINE_* flags */
138     usb_audio_format_t fmt;
139     uint64_t        af_defgain;

141     unsigned        intrate;   /* interrupt rate */
142     unsigned        sampsz;    /* sample size */
143     unsigned        framesz;   /* frame size */
144     unsigned        fragsz;    /* fragment size */
145     unsigned        nfrags;    /* number of fragments in buffer */
146     unsigned        fragfr;    /* number of frames per fragment */
147     unsigned        frmsshift; /* right shift: frames in sample cnt */
148     unsigned        smszshift; /* left shift: sample cnt * sampsz */

151     caddr_t         bufp;      /* I/O buf; framework to/from drv */
152     unsigned        bufpsz;   /* buffer size */
153     caddr_t         bufpos;    /* buffer position */
154     caddr_t         bufendp;   /* end of buffer */

157     uint64_t        frames;
158     uint64_t        io_count;  /* i/o requests from the driver */
159     uint64_t        bufio_count; /* i/o requests to the framework */

161     boolean_t       started;
162     boolean_t       busy;

164     kcondvar_t      usb_audio_cv;

166     kmutex_t        lock;
167 } usb_audio_eng_t;

170 /* limits */
171 #define USB_AC_MAX_PLUMBED      3      /* play, record, hid */
172 #define USB_AC_MAX_AS_PLUMBED  2      /* play, record */
173 typedef struct usb_ac_state    usb_ac_state_t;
174 typedef struct usb_audio_ctrl {
175     audio_ctrl_t      *af_ctrlp; /* framework handle */
176     usb_ac_state_t    *statep;

178     kmutex_t          ctrl_mutex;
179     uint64_t          cval;      /* current control value */
180 } usb_audio_ctrl_t;

182 enum {
183     CTL_VOLUME_MONO = 0,
184     CTL_VOLUME_STERO,
185     CTL_REC_MONO,
186     CTL_REC_STERO,
187     CTL_REC_SRC,
188     CTL_MONITOR_GAIN,
189     CTL_MIC_BOOST,
190     CTL_NUM
191 };

193 #define USB_AC_ENG_MAX  2

```

```

195 /* usb_ac soft state */
196 struct usb_ac_state {

198     dev_info_t      *usb_ac_dip;
199     uint_t          usb_ac_instance;
200     usb_log_handle_t usb_log_handle;

202     uint_t          usb_ac_dev_state;
203     uint_t          usb_ac_ifno;
204     kmutex_t        usb_ac_mutex;

206     usb_client_dev_data_t *usb_ac_dev_data; /* registration data */
207     audio_dev_t      *usb_ac_audio_dev;

212     usb_audio_eng_t  engines[USB_AC_ENG_MAX];

216     int             flags;
217     usb_audio_ctrl_t *controls[CTL_NUM];

219     /* descriptors */
220     usb_if_descr_t  usb_ac_if_descr;

222     /* unit number array, indexed by unit ID */
223     uint_t          usb_ac_max_unit;
224     usb_ac_unit_list_t *usb_ac_units;

226     /* adjacency matrix for reflecting connections */
227     uchar_t         *usb_ac_connections;
228     size_t          usb_ac_connections_len;
229     uchar_t         *usb_ac_connections_a;
230     size_t          usb_ac_connections_a_len;
231     uchar_t         *usb_ac_unit_type;
232     uchar_t         *usb_ac_traverse_path;
233     uchar_t         usb_ac_traverse_path_index;

235     /* port types, eg LINE IN, Micr, Speakers */
236     uint64_t        usb_ac_input_ports;
237     uint64_t        usb_ac_output_ports;

239     /* pipe handle */
240     usb_pipe_handle_t usb_ac_default_ph;

242     /* serial access */
243     usb_serialization_t usb_ac_ser_acc;

245     /* power management */
246     usb_ac_power_t   *usb_ac_pm; /* power capabilities */

248     /* mixer registration data */
249     uint_t          usb_ac_registered_with_mixer;

251     /* plumbing management */
252     uint_t          usb_ac_plumbing_state;
253     ushort_t        usb_ac_busy_count;
254     usb_ac_plumbed_t usb_ac_plumbed[USB_AC_MAX_PLUMBED];

256     /* Current plumbed module index to usb_ac_plumbed structure */
257     int             usb_ac_current_plumbed_index;

259     /* per streams interface info */

```

```

260     usb_ac_streams_info_t    usb_ac_streams[USB_AC_MAX_AS_PLUMBED];

263     ddi_taskq_t              *tqp;

265     char                     dstr[64];
266 };

268 /* warlock directives, stable data */
269 _NOTE(MUTEX_PROTECTS_DATA(usb_ac_state_t::usb_ac_mutex, usb_ac_state_t))
270 _NOTE(MUTEX_PROTECTS_DATA(usb_ac_state_t::usb_ac_mutex, usb_ac_power_t))
271 _NOTE(MUTEX_PROTECTS_DATA(usb_ac_state_t::usb_ac_mutex, usb_ac_plumbed_t))
272 _NOTE(MUTEX_PROTECTS_DATA(usb_audio_eng_t::lock, usb_audio_eng_t))
273 _NOTE(MUTEX_PROTECTS_DATA(usb_audio_eng_t::lock, usb_audio_format_t))
274 _NOTE(MUTEX_PROTECTS_DATA(usb_audio_ctrl_t::ctrl_mutex, usb_audio_ctrl_t))

277 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_dip))
278 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_ser_acc))
279 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_pm))
280 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_instance))
281 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_default_ph))
282 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_log_handle))
283 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_if_descr))
284 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_dev_data))
285 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_ifno))
286 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::flags))
287 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_input_ports))
288 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::engines))
289 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::usb_ac_audio_dev))
290 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_state_t::controls))

292 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::af_eflags))
293 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::streams))
294 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::statep))
295 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::fmt))
296 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::fragfr))
297 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::frsmshift))
298 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::started))
299 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::af_engp))
300 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::io_count))
301 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_eng_t::intrate))

303 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_ctrl_t::statep))
304 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_ctrl_t::af_ctrlp))
305 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_ctrl_t::cval))

307 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_plumbed_t::acp_tqp))
308 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_ac_plumbed_t::acp_uacp))

310 _NOTE(DATA_READABLE_WITHOUT_LOCK(usb_audio_format_t::ch))

312 /* usb_ac driver only care about two states:  plumbed or unplumbed */
313 #define USB_AC_STATE_UNPLUMBED      0
314 #define USB_AC_STATE_PLUMBED       1
315 #define USB_AC_STATE_PLUMBED_RESTORING  2

317 /* Default pipe states */
318 #define USB_AC_DEF_CLOSED           0
319 #define USB_AC_DEF_OPENED          1

321 #define USB_AC_BUFFER_SIZE          256    /* descriptor buffer size */

324 /*
325  * delay before restoring state

```

```

326  */
327 #define USB_AC_RESTORE_DELAY        drv_usecstohz(1000000)

329 /* value for acp_driver */
330 #define USB_AS_PLUMBED              1
331 #define USB_AH_PLUMBED              2
332 #define UNKNOWN_PLUMBED            3

334 #define AF_REGISTERED                0x1
335 #define AD_SETUP                     0x10

338 int usb_audio_attach(usb_ac_state_t *);
339 /*
340  * framework gain range
341  */
342 #define AUDIO_CTRL_STEREO_VAL(l, r)  (((l) & 0xff) | (((r) & 0xff) << 8))
343 #define AUDIO_CTRL_STEREO_LEFT(v)   ((uint8_t)((v) & 0xff))
344 #define AUDIO_CTRL_STEREO_RIGHT(v)  ((uint8_t)((v) >> 8) & 0xff)

347 #define AF_MAX_GAIN                  100
348 #define AF_MIN_GAIN                 0

352 int usb_ac_get_audio(void *, void *, int);

354 void usb_ac_send_audio(void *, void *, int);

356 void usb_ac_stop_play(usb_ac_state_t *, usb_audio_eng_t *);

359 #ifdef __cplusplus
360 }

```

_____ 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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 #ifndef _SYS_USB_USBVC_VAR_H
27 #define _SYS_USB_USBVC_VAR_H

30 #ifdef __cplusplus
31 extern "C" {
32 #endif

34 #include <sys/list.h>
35 #include <sys/sysmacros.h>
36 #endif /* ! codereview */
37 #include <sys/usb/usba/usbai_private.h>
38 #include <sys/videodev2.h>
39 #include <sys/usb/clients/video/usbvc/usbvc.h>

41 typedef struct usbvc_state usbvc_state_t;

43 /*
44  * Power Management support
45  */
46 typedef struct usbvc_power {

48     void            *usbvc_state; /* points back to usbvc_state */
49     uint8_t         usbvc_pwr_states; /* bit mask of device pwr states */
50     int             usbvc_pm_busy;

52     /* Wakeup and power transistion capabilities of an interface */
53     uint8_t         usbvc_pm_capabilities;

55     /* flag to indicate if driver is about to raise power level */
56     boolean_t       usbvc_raise_power;

58     uint8_t         usbvc_current_power;
59     uint8_t         usbvc_wakeup_enabled;
60 } usbvc_power_t;

```

```

62 /* Raw data buf from the USB cam */
63 typedef struct usbvc_buf
64 {
65     uchar_t *data;
66     uint_t len; /* the length of the allocated memory of data */
67     uint_t filled; /* number of bytes filled */
68     uint_t len_read; /* bytes read */
69     uchar_t status; /* empty, filling done, read done */

71     /* cookie used for memory mapping */
72     ddi_umem_cookie_t umem_cookie;
73     struct v4l2_buffer v4l2_buf;
74     list_node_t buf_node; /* list */
75 } usbvc_buf_t;

77 /* Group data buf related lists and other elements */
78 typedef struct usbvc_buf_grp
79 {
80     list_t uv_buf_free;
81     list_t uv_buf_done;
82     usbvc_buf_t *buf_filling;
83     uint_t buf_cnt;
84     usbvc_buf_t *buf_head;
85 } usbvc_buf_grp_t;

87 /*
88  * UVC Spec: one format descriptor may be followed by several frame
89  * descriptors, one still image descriptor and one color matching descriptor.
90  * It is called a format group. There might be several format groups follow
91  * one input/output header.
92  */
93 typedef struct usbvc_format_group {
94     usbvc_format_descr_t *format;
95     usbvc_frames_t *frames;
96     uint8_t frame_cnt;

98     /* bytes per pix, used to calculate bytesperline */
99     uint8_t v4l2_bpp;

101     uint8_t v4l2_color;
102     uint32_t v4l2_pixelformat; /* fcc, pixelformat */
103     usbvc_still_image_frame_t *still;
104     usbvc_color_matching_descr_t *color;
105     usbvc_frames_t *cur_frame;
106 } usbvc_format_group_t;

108 /* A stream interface may have several format groups */
109 typedef struct usbvc_stream_if {

111     /* The actual format groups we parsed for the stream interface */
112     uint8_t fmtgrp_cnt;

114     usb_if_data_t *if_descr;
115     usbvc_input_header_t *input_header;
116     usbvc_output_header_t *output_header;
117     usbvc_format_group_t *format_group;
118     usbvc_format_group_t *cur_format_group;
119     usbvc_vs_probe_commit_t ctrl_pc;
120     usb_ep_descr_t *curr_ep; /* current isoc ep descr */
121     usb_pipe_handle_t datain_ph; /* current isoc pipe handle */
122     uint_t curr_alt; /* current alternate */

124     /* The max payload that the isoc data EPs can support */
125     uint32_t max_isoc_payload;

127     uchar_t start_polling; /* indicate if isoc polling started */

```

```

129  /*
130  * To flag if VIDIIOC_STREAMON is executed, only used by STREAM mode
131  * for suspend/resume. If it's non-zero, we'll have to resume the
132  * device's isoc polling operation after resume.
133  */
134  uint8_t      stream_on;

136  uchar_t      fid;          /* the MJPEG FID bit */
137  usbvc_buf_grp_t buf_read; /* buf used for read I/O */
138  uint8_t      buf_read_num; /* desired buf num for read I/O */
139  usbvc_buf_grp_t buf_map;  /* buf used for mmap I/O */
140  list_node_t  stream_if_node;
141 } usbvc_stream_if_t;

143 /* video interface collection */
144 typedef struct usbvc_vic {

146     /* bFirstInterface, the video control interface num of this VIC */
147     uint8_t vctrl_if_num;

149     /*
150     * bInterfaceCount -1, the total number of stream interfaces
151     * belong to this VIC
152     */
153     uint8_t vstrm_if_cnt;
154 } usbvc_vic_t;

156 /* Macros */
157 #define USBVC_OPEN          0x00000001

159 /* For serialization. */
160 #define USBVC_SER_NOSIG B_FALSE
161 #define USBVC_SER_SIG   B_TRUE

163 /*
164  * Masks for debug printing
165  */
166 #define PRINT_MASK_ATT A      0x00000001
167 #define PRINT_MASK_OPEN 0x00000002
168 #define PRINT_MASK_CLOSE 0x00000004
169 #define PRINT_MASK_READ  0x00000008
170 #define PRINT_MASK_IOCTL 0x00000010
171 #define PRINT_MASK_PM    0x00000020
172 #define PRINT_MASK_CB    0x00000040
173 #define PRINT_MASK_HOTPLUG 0x00000080
174 #define PRINT_MASK_DEVCTRL 0x00000100
175 #define PRINT_MASK_DEVMAP 0x00000200
176 #define PRINT_MASK_ALL    0xFFFFFFFF

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

178 #define USBVC_MAX_PKTS 40

180 #define USBVC_DEFAULT_READ_BUF_NUM 3
181 #define USBVC_MAX_READ_BUF_NUM 40
182 #define USBVC_MAX_MAP_BUF_NUM 40

184 /* According to UVC specs, the frame interval is in 100ns unit */
185 #define USBVC_FRAME_INTERVAL_DENOMINATOR 1000000

187 /* Only D3..D0 are writable, Table 4-6, UVC Spec */
188 #define USBVC_POWER_MODE_MASK 0xf0;

190 enum usbvc_buf_status {
191     USBVC_BUF_INIT          = 0, /* Allocated, to be queued */

```

```

192     USBVC_BUF_MAPPED      = 1, /* For map I/O only. Memory is mapped. */
193     USBVC_BUF_EMPTY      = 2, /* not initialized, to be filled */

195     /*
196     * buf is filled with a full frame without any errors,
197     * it will be moved to full list.
198     */
199     USBVC_BUF_DONE          = 4,

201     /*
202     * buf is filled to full but no EOF bit is found at the end
203     * of video data
204     */
205     USBVC_BUF_ERR          = 8
206 };
_____unchanged_portion_omitted_

```

new/usr/src/uts/common/xen/io/xnb.h

1

```
*****
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>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23 * Copyright 2010 Sun Microsystems, Inc. All rights reserved.
24 * Use is subject to license terms.
25 *
26 * xnb.h - definitions for Xen dom0 network driver
27 */

29 #ifndef _SYS_XNB_H
30 #define _SYS_XNB_H

32 #include <sys/types.h>
33 #include <sys/kstat.h>
34 #include <sys/stream.h>
35 #include <sys/ethernet.h>
36 #include <sys/hypervisor.h>
37 #include <sys/sysmacros.h>
38 #endif /* ! codereview */
39 #include <xen/public/io/netif.h>

41 #ifdef __cplusplus
42 extern "C" {
43 #endif

45 #define NET_TX_RING_SIZE    __CONST_RING_SIZE(netif_tx, PAGESIZE)
46 #define NET_RX_RING_SIZE   __CONST_RING_SIZE(netif_rx, PAGESIZE)
47 #define NET_TX_RING_SIZE   __RING_SIZE((netif_tx_sring_t *)0, PAGESIZE)
48 #define NET_RX_RING_SIZE   __RING_SIZE((netif_rx_sring_t *)0, PAGESIZE)

48 #define XNBMAXPKT          1500          /* MTU size */

50 /* DEBUG flags */
51 #define XNBDDI              0x01
52 #define XNBTRACE            0x02
53 #define XNBSEND             0x04
54 #define XNBRECV             0x08
55 #define XNBINTR             0x10
56 #define XNBRING             0x20
57 #define XNBCKSUM            0x40
```

new/usr/src/uts/common/xen/io/xnb.h

2

```
59 #define XNB_STATE_INIT    0x01
60 #define XNB_STATE_READY   0x02

62 typedef struct xnb xnb_t;

64 /*
65 * The xnb module provides core inter-domain network protocol functionality.
66 * It is connected to the rest of Solaris in two ways:
67 * - as a GLDv3 driver (with xnbu),
68 * - as a GLDv3 consumer (with xnb).
69 *
70 * The different modes of operation are termed "flavours" and each
71 * instance of an xnb based driver operates in one and only one mode.
72 * The common xnb driver exports a set of functions to these drivers
73 * (declarations at the foot of this file) and calls back into the
74 * drivers via the xnb_flavour_t structure.
75 */
76 typedef struct xnb_flavour {
77     void          (*xf_from_peer)(xnb_t *, mblk_t *);
78     boolean_t     (*xf_peer_connected)(xnb_t *);
79     void          (*xf_peer_disconnected)(xnb_t *);
80     boolean_t     (*xf_hotplug_connected)(xnb_t *);
81     boolean_t     (*xf_start_connect)(xnb_t *);
82     mblk_t        (*xf_cksum_from_peer)(xnb_t *, mblk_t *, uint16_t);
83     uint16_t      (*xf_cksum_to_peer)(xnb_t *, mblk_t *);
84     boolean_t     (*xf_mcast_add)(xnb_t *, ether_addr_t *);
85     boolean_t     (*xf_mcast_del)(xnb_t *, ether_addr_t *);
86 } xnb_flavour_t;
    unchanged portion omitted
```

new/usr/src/uts/common/xen/io/xfn.h

1

```
*****
4750 Thu Feb 25 15:39:44 2016
new/usr/src/uts/common/xen/io/xfn.h
3373 gcc >= 4.5 concerns about offsetof()
Portions contributed by: Igor Pashev <pashev.igor@gmail.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */

27 #ifndef _SYS_XNF_H
28 #define _SYS_XNF_H

30 #ifdef __cplusplus
31 extern "C" {
32 #endif

34 #define NET_TX_RING_SIZE    _CONST_RING_SIZE(netif_tx, PAGESIZE)
35 #define NET_RX_RING_SIZE    _CONST_RING_SIZE(netif_rx, PAGESIZE)
34 #define NET_TX_RING_SIZE    _RING_SIZE((netif_tx_sring_t *)0, PAGESIZE)
35 #define NET_RX_RING_SIZE    _RING_SIZE((netif_rx_sring_t *)0, PAGESIZE)

37 #define XNF_MAXPKT          1500          /* MTU size */
38 #define XNF_FRAME_SIZE      1514          /* frame size including MAC header */

40 /* DEBUG flags */
41 #define XNF_DEBUG_DDI        0x01
42 #define XNF_DEBUG_TRACE     0x02

44 /*
45  * Information about each receive buffer and any transmit look-aside
46  * buffers.
47  */
48 typedef struct xnf_buf {
49     frtn_t          free_rtn;
50     struct xnf      *xnfp;
51     ddi_dma_handle_t dma_handle;
52     caddr_t         buf;          /* DMA-able data buffer */
53     paddr_t         buf_phys;
54     mfn_t           buf_mfn;
55     size_t          len;
56     struct xnf_buf *next; /* For linking into free list */
57     ddi_acc_handle_t acc_handle;
58     grant_ref_t     grant_ref; /* grant table reference */

```

new/usr/src/uts/common/xen/io/xfn.h

2

```
59     uint16_t        id;          /* buffer id */
60     unsigned int    gen;
61 } 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>
*****
1 /*****
2  * ring.h
3  *
4  * Shared producer-consumer ring macros.
5  *
6  * Permission is hereby granted, free of charge, to any person obtaining a copy
7  * of this software and associated documentation files (the "Software"), to
8  * deal in the Software without restriction, including without limitation the
9  * rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
10 * sell copies of the Software, and to permit persons to whom the Software is
11 * furnished to do so, subject to the following conditions:
12 *
13 * The above copyright notice and this permission notice shall be included in
14 * all copies or substantial portions of the Software.
15 *
16 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
17 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
18 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
19 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
20 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
21 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
22 * DEALINGS IN THE SOFTWARE.
23 *
24 * Tim Deegan and Andrew Warfield November 2004.
25 */

27 #ifndef __XEN_PUBLIC_IO_RING_H__
28 #define __XEN_PUBLIC_IO_RING_H__

30 #include "../xen-compat.h"

32 #if __XEN_INTERFACE_VERSION__ < 0x00030208
33 #define xen_mb() mb()
34 #define xen_rmb() rmb()
35 #define xen_wmb() wmb()
36 #endif

38 typedef unsigned int RING_IDX;

40 /* Round a 32-bit unsigned constant down to the nearest power of two. */
41 #define __RD2(_x) (((_x) & 0x00000002) ? 0x2 : ((_x) & 0x1))
42 #define __RD4(_x) (((_x) & 0x0000000c) ? __RD2((_x)>>2)<<2 : __RD2(_x))
43 #define __RD8(_x) (((_x) & 0x000000f0) ? __RD4((_x)>>4)<<4 : __RD4(_x))
44 #define __RD16(_x) (((_x) & 0x0000fff0) ? __RD8((_x)>>8)<<8 : __RD8(_x))
45 #define __RD32(_x) (((_x) & 0xfffff000) ? __RD16((_x)>>16)<<16 : __RD16(_x))

47 /*
48  * Calculate size of a shared ring, given the total available space for the
49  * ring and indexes (_sz), and the name tag of the request/response structure.
50  * A ring contains as many entries as will fit, rounded down to the nearest
51  * power of two (so we can mask with (size-1) to loop around).
52  */
53 #define __CONST_RING_SIZE(_s, _sz) \
54     (__RD32(((_sz) - offsetof(struct _s##_sring, ring)) / \
55             sizeof((struct _s##_sring *)0)->ring[0])))
56 /*
57  * The same for passing in an actual pointer instead of a name tag.
58  */
59 #endif /* ! codereview */
60 #define __RING_SIZE(_s, _sz) \

```

```

61     (__RD32(((_sz) - (long)(_s)->ring + (long)(_s)) / sizeof((_)>ring[0])))
63 /*
64  * Macros to make the correct C datatypes for a new kind of ring.
65  *
66  * To make a new ring datatype, you need to have two message structures,
67  * let's say request_t, and response_t already defined.
68  *
69  * In a header where you want the ring datatype declared, you then do:
70  *
71  *     DEFINE_RING_TYPES(mytag, request_t, response_t);
72  *
73  * These expand out to give you a set of types, as you can see below.
74  * The most important of these are:
75  *
76  *     mytag_sring_t - The shared ring.
77  *     mytag_front_ring_t - The 'front' half of the ring.
78  *     mytag_back_ring_t - The 'back' half of the ring.
79  *
80  * To initialize a ring in your code you need to know the location and size
81  * of the shared memory area (PAGE_SIZE, for instance). To initialise
82  * the front half:
83  *
84  *     mytag_front_ring_t front_ring;
85  *     SHARED_RING_INIT(&mytag_sring_t *)shared_page);
86  *     FRONT_RING_INIT(&front_ring, (mytag_sring_t *)shared_page, PAGE_SIZE);
87  *
88  * Initializing the back follows similarly (note that only the front
89  * initializes the shared ring):
90  *
91  *     mytag_back_ring_t back_ring;
92  *     BACK_RING_INIT(&back_ring, (mytag_sring_t *)shared_page, PAGE_SIZE);
93  */

95 #define DEFINE_RING_TYPES(_name, _req_t, _rsp_t) \
96 \
97 /* Shared ring entry */ \
98 union __name##_sring_entry { \
99     _req_t req; \
100    _rsp_t rsp; \
101 }; \
102 \
103 /* Shared ring page */ \
104 struct __name##_sring { \
105     RING_IDX req_prod, req_event; \
106     RING_IDX rsp_prod, rsp_event; \
107     uint8_t pad[48]; \
108     union __name##_sring_entry ring[1]; /* variable-length */ \
109 }; \
110 \
111 /* "Front" end's private variables */ \
112 struct __name##_front_ring { \
113     RING_IDX req_prod_pvt; \
114     RING_IDX rsp_cons; \
115     unsigned int nr_ents; \
116     struct __name##_sring *sring; \
117 }; \
118 \
119 /* "Back" end's private variables */ \
120 struct __name##_back_ring { \
121     RING_IDX rsp_prod_pvt; \
122     RING_IDX req_cons; \
123     unsigned int nr_ents; \
124     struct __name##_sring *sring; \
125 }; \
126 \

```

```

127 /* Syntactic sugar */
128 typedef struct __name##_sring __name##_sring_t;
129 typedef struct __name##_front_ring __name##_front_ring_t;
130 typedef struct __name##_back_ring __name##_back_ring_t
131
132 /*
133  * Macros for manipulating rings.
134  *
135  * FRONT_RING_whatever works on the "front end" of a ring: here
136  * requests are pushed on to the ring and responses taken off it.
137  *
138  * BACK_RING_whatever works on the "back end" of a ring: here
139  * requests are taken off the ring and responses put on.
140  *
141  * N.B. these macros do NO INTERLOCKS OR FLOW CONTROL.
142  * This is OK in 1-for-1 request-response situations where the
143  * requestor (front end) never has more than RING_SIZE()-1
144  * outstanding requests.
145  */
146
147 /* Initialising empty rings */
148 #define SHARED_RING_INIT(_s) do {
149     (_s)->req_prod = (_s)->rsp_prod = 0;
150     (_s)->req_event = (_s)->rsp_event = 1;
151     (void)memset((_s)->pad, 0, sizeof((_s)->pad));
152 } while(0)
153
154 #define FRONT_RING_INIT(_r, _s, __size) do {
155     (_r)->req_prod_pvt = 0;
156     (_r)->rsp_cons = 0;
157     (_r)->nr_ents = __RING_SIZE(_s, __size);
158     (_r)->sring = (_s);
159 } while (0)
160
161 #define BACK_RING_INIT(_r, _s, __size) do {
162     (_r)->rsp_prod_pvt = 0;
163     (_r)->req_cons = 0;
164     (_r)->nr_ents = __RING_SIZE(_s, __size);
165     (_r)->sring = (_s);
166 } while (0)
167
168 /* Initialize to existing shared indexes -- for recovery */
169 #define FRONT_RING_ATTACH(_r, _s, __size) do {
170     (_r)->sring = (_s);
171     (_r)->req_prod_pvt = (_s)->req_prod;
172     (_r)->rsp_cons = (_s)->rsp_prod;
173     (_r)->nr_ents = __RING_SIZE(_s, __size);
174 } while (0)
175
176 #define BACK_RING_ATTACH(_r, _s, __size) do {
177     (_r)->sring = (_s);
178     (_r)->rsp_prod_pvt = (_s)->rsp_prod;
179     (_r)->req_cons = (_s)->req_prod;
180     (_r)->nr_ents = __RING_SIZE(_s, __size);
181 } while (0)
182
183 /* How big is this ring? */
184 #define RING_SIZE(_r)
185     ((_r)->nr_ents)
186
187 /* Number of free requests (for use on front side only). */
188 #define RING_FREE_REQUESTS(_r)
189     (RING_SIZE(_r) - ((_r)->req_prod_pvt - (_r)->rsp_cons))
190
191 /* Test if there is an empty slot available on the front ring.
192  * (This is only meaningful from the front. )

```

```

193 */
194 #define RING_FULL(_r)
195     (RING_FREE_REQUESTS(_r) == 0)
196
197 /* Test if there are outstanding messages to be processed on a ring. */
198 #define RING_HAS_UNCONSUMED_RESPONSES(_r)
199     ((_r)->sring->rsp_prod - (_r)->rsp_cons)
200
201 #ifdef __GNUC__
202 #define RING_HAS_UNCONSUMED_REQUESTS(_r) ({
203     unsigned int req = (_r)->sring->req_prod - (_r)->req_cons;
204     unsigned int rsp = RING_SIZE(_r) -
205         ((_r)->req_cons - (_r)->rsp_prod_pvt);
206     req < rsp ? req : rsp;
207 })
208 #else
209 /* Same as above, but without the nice GCC ({ ... }) syntax. */
210 #define RING_HAS_UNCONSUMED_REQUESTS(_r)
211     ((((_r)->sring->req_prod - (_r)->req_cons) <
212     (RING_SIZE(_r) - ((_r)->req_cons - (_r)->rsp_prod_pvt))) ?
213     ((_r)->sring->req_prod - (_r)->req_cons) :
214     (RING_SIZE(_r) - ((_r)->req_cons - (_r)->rsp_prod_pvt)))
215 #endif
216
217 /* Direct access to individual ring elements, by index. */
218 #define RING_GET_REQUEST(_r, _idx)
219     (&((_r)->sring->ring[(_idx) & (RING_SIZE(_r) - 1)].req))
220
221 #define RING_GET_RESPONSE(_r, _idx)
222     (&((_r)->sring->ring[(_idx) & (RING_SIZE(_r) - 1)].rsp))
223
224 /* Loop termination condition: Would the specified index overflow the ring? */
225 #define RING_REQUEST_CONS_OVERFLOW(_r, _cons)
226     (((_cons) - (_r)->rsp_prod_pvt) >= RING_SIZE(_r))
227
228 #define RING_PUSH_REQUESTS(_r) do {
229     xen_wmb(); /* back sees requests /before/ updated producer index */
230     (_r)->sring->req_prod = (_r)->req_prod_pvt;
231 } while (0)
232
233 #define RING_PUSH_RESPONSES(_r) do {
234     xen_wmb(); /* front sees resps /before/ updated producer index */
235     (_r)->sring->rsp_prod = (_r)->rsp_prod_pvt;
236 } while (0)
237
238 /*
239  * Notification hold-off (req_event and rsp_event):
240  *
241  * When queueing requests or responses on a shared ring, it may not always be
242  * necessary to notify the remote end. For example, if requests are in flight
243  * in a backend, the front may be able to queue further requests without
244  * notifying the back (if the back checks for new requests when it queues
245  * responses).
246  *
247  * When enqueueing requests or responses:
248  *
249  * Use RING_PUSH_{REQUESTS,RESPONSES}_AND_CHECK_NOTIFY(). The second argument
250  * is a boolean return value. True indicates that the receiver requires an
251  * asynchronous notification.
252  *
253  * After dequeuing requests or responses (before sleeping the connection):
254  *
255  * Use RING_FINAL_CHECK_FOR_REQUESTS() or RING_FINAL_CHECK_FOR_RESPONSES().
256  * The second argument is a boolean return value. True indicates that there
257  * are pending messages on the ring (i.e., the connection should not be put
258  * to sleep).

```

```

259 *
260 * These macros will set the req_event/rsp_event field to trigger a
261 * notification on the very next message that is enqueued. If you want to
262 * create batches of work (i.e., only receive a notification after several
263 * messages have been enqueued) then you will need to create a customised
264 * version of the FINAL_CHECK macro in your own code, which sets the event
265 * field appropriately.
266 */

268 #define RING_PUSH_REQUESTS_AND_CHECK_NOTIFY(_r, _notify) do { \
269     RING_IDX __old = (_r)->sring->req_prod; \
270     RING_IDX __new = (_r)->req_prod_pvt; \
271     xen_wmb(); /* back sees requests /before/ updated producer index */ \
272     (_r)->sring->req_prod = __new; \
273     xen_mb(); /* back sees new requests /before/ we check req_event */ \
274     (_notify) = ((RING_IDX)(__new - (_r)->sring->req_event) < \
275                 (RING_IDX)(__new - __old)); \
276 } while (0)

278 #define RING_PUSH_RESPONSES_AND_CHECK_NOTIFY(_r, _notify) do { \
279     RING_IDX __old = (_r)->sring->rsp_prod; \
280     RING_IDX __new = (_r)->rsp_prod_pvt; \
281     xen_wmb(); /* front sees resps /before/ updated producer index */ \
282     (_r)->sring->rsp_prod = __new; \
283     xen_mb(); /* front sees new resps /before/ we check rsp_event */ \
284     (_notify) = ((RING_IDX)(__new - (_r)->sring->rsp_event) < \
285                 (RING_IDX)(__new - __old)); \
286 } while (0)

288 #define RING_FINAL_CHECK_FOR_REQUESTS(_r, _work_to_do) do { \
289     (_work_to_do) = RING_HAS_UNCONSUMED_REQUESTS(_r); \
290     if (_work_to_do) break; \
291     (_r)->sring->req_event = (_r)->req_cons + 1; \
292     xen_mb(); \
293     (_work_to_do) = RING_HAS_UNCONSUMED_REQUESTS(_r); \
294 } while (0)

296 #define RING_FINAL_CHECK_FOR_RESPONSES(_r, _work_to_do) do { \
297     (_work_to_do) = RING_HAS_UNCONSUMED_RESPONSES(_r); \
298     if (_work_to_do) break; \
299     (_r)->sring->rsp_event = (_r)->rsp_cons + 1; \
300     xen_mb(); \
301     (_work_to_do) = RING_HAS_UNCONSUMED_RESPONSES(_r); \
302 } while (0)

304 #endif /* __XEN_PUBLIC_IO_RING_H__ */

306 /*
307 * Local variables:
308 * mode: C
309 * c-set-style: "BSD"
310 * c-basic-offset: 4
311 * tab-width: 4
312 * indent-tabs-mode: nil
313 * End:
314 */

```

```

*****
194604 Thu Feb 25 15:39:45 2016
new/usr/src/uts/sun/io/scsi/adapters/sf.c
2976 remove useless offsetof() macros
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 * Copyright (c) 2011 Bayard G. Bell. All rights reserved.
25 */

27 /*
28 * sf - Solaris Fibre Channel driver
29 *
30 * This module implements some of the Fibre Channel FC-4 layer, converting
31 * from FC frames to SCSI and back. (Note: no sequence management is done
32 * here, though.)
33 */

35 #if defined(lint) && !defined(DEBUG)
36 #define DEBUG 1
37 #endif

39 /*
40 * XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
41 * Need to use the ugly RAID LUN mappings in FCP Annex D
42 * to prevent SCSI from barfing. This *REALLY* needs to
43 * be addressed by the standards committee.
44 */
45 #define RAID_LUNS 1

47 #ifdef DEBUG
48 static int sfdebug = 0;
49 #include <sys/debug.h>

51 #define SF_DEBUG(level, args) \
52     if (sfdebug >= (level)) sf_log args
53 #else
54 #define SF_DEBUG(level, args)
55 #endif

57 static int sf_bus_config_debug = 0;

59 /* Why do I have to do this? */
60 #define offsetof(s, m) (size_t)&(((s *)0)->m)

```

```

59 #include <sys/scsi/scsi.h>
60 #include <sys/fc4/fcal.h>
61 #include <sys/fc4/fcp.h>
62 #include <sys/fc4/fcal_linkapp.h>
63 #include <sys/socal_cq_defs.h>
64 #include <sys/fc4/fcal_transport.h>
65 #include <sys/fc4/fcio.h>
66 #include <sys/scsi/adapters/sfvar.h>
67 #include <sys/scsi/impl/scsi_reset_notify.h>
68 #include <sys/stat.h>
69 #include <sys/varargs.h>
70 #include <sys/var.h>
71 #include <sys/thread.h>
72 #include <sys/proc.h>
73 #include <sys/kstat.h>
74 #include <sys/devctl.h>
75 #include <sys/scsi/targets/ses.h>
76 #include <sys/callb.h>
77 #include <sys/sysmacros.h>
78 #endif /* ! codereview */

80 static int sf_info(dev_info_t *, ddi_info_cmd_t, void *, void **);
81 static int sf_attach(dev_info_t *, ddi_attach_cmd_t);
82 static int sf_detach(dev_info_t *, ddi_detach_cmd_t);
83 static void sf_softstate_unlink(struct sf *);
84 static int sf_scsi_bus_config(dev_info_t *parent, uint_t flag,
85     ddi_bus_config_op_t op, void *arg, dev_info_t **childp);
86 static int sf_scsi_bus_unconfig(dev_info_t *parent, uint_t flag,
87     ddi_bus_config_op_t op, void *arg);
88 static int sf_scsi_tgt_init(dev_info_t *, dev_info_t *,
89     scsi_hba_tran_t *, struct scsi_device *);
90 static void sf_scsi_tgt_free(dev_info_t *, dev_info_t *,
91     scsi_hba_tran_t *, struct scsi_device *);
92 static int sf_pkt_alloc_extern(struct sf *, struct sf_pkt *,
93     int, int, int);
94 static void sf_pkt_destroy_extern(struct sf *, struct sf_pkt *);
95 static struct scsi_pkt *sf_scsi_init_pkt(struct scsi_address *,
96     struct scsi_pkt *, struct buf *, int, int, int, int (*), caddr_t);
97 static void sf_scsi_destroy_pkt(struct scsi_address *, struct scsi_pkt *);
98 static void sf_scsi_dmafree(struct scsi_address *, struct scsi_pkt *);
99 static void sf_scsi_sync_pkt(struct scsi_address *, struct scsi_pkt *);
100 static int sf_scsi_reset_notify(struct scsi_address *, int,
101     void (*)(caddr_t), caddr_t);
102 static int sf_scsi_get_name(struct scsi_device *, char *, int);
103 static int sf_scsi_get_bus_addr(struct scsi_device *, char *, int);
104 static int sf_add_cr_pool(struct sf *);
105 static int sf_cr_alloc(struct sf *, struct sf_pkt *, int (*));
106 static void sf_cr_free(struct sf_cr_pool *, struct sf_pkt *);
107 static void sf_crpool_free(struct sf *);
108 static int sf_kmem_cache_constructor(void *, void *, int);
109 static void sf_kmem_cache_destructor(void *, void *);
110 static void sf_statec_callback(void *, int);
111 static int sf_login(struct sf *, uchar_t, uchar_t, uint_t, int);
112 static int sf_els_transport(struct sf *, struct sf_els_hdr *);
113 static void sf_els_callback(struct fc_lun_packet *);
114 static int sf_do_prli(struct sf *, struct sf_els_hdr *, struct la_els_logi *);
115 static int sf_do_adisc(struct sf *, struct sf_els_hdr *);
116 static int sf_do_reportlun(struct sf *, struct sf_els_hdr *,
117     struct sf_target *);
118 static void sf_reportlun_callback(struct fc_lun_packet *);
119 static int sf_do_inquiry(struct sf *, struct sf_els_hdr *,
120     struct sf_target *);
121 static void sf_inq_callback(struct fc_lun_packet *);
122 static struct fc_lun_packet *sf_els_alloc(struct sf *, uchar_t, int, int,
123     int, caddr_t *, caddr_t *);
124 static void sf_els_free(struct fc_lun_packet *);

```

```

125 static struct sf_target *sf_create_target(struct sf *,
126     struct sf_els_hdr *, int, int64_t);
127 #ifdef RAID_LUNS
128 static struct sf_target *sf_lookup_target(struct sf *, uchar_t *, int);
129 #else
130 static struct sf_target *sf_lookup_target(struct sf *, uchar_t *, int64_t);
131 #endif
132 static void sf_finish_init(struct sf *, int);
133 static void sf_offline_target(struct sf *, struct sf_target *);
134 static void sf_create_devinfo(struct sf *, struct sf_target *, int);
135 static int sf_create_props(dev_info_t *, struct sf_target *, int);
136 static int sf_commoncap(struct scsi_address *, char *, int, int, int);
137 static int sf_getcap(struct scsi_address *, char *, int);
138 static int sf_setcap(struct scsi_address *, char *, int, int);
139 static int sf_abort(struct scsi_address *, struct scsi_pkt *);
140 static int sf_reset(struct scsi_address *, int);
141 static void sf_abort_all(struct sf *, struct sf_target *, int, int, int);
142 static int sf_start(struct scsi_address *, struct scsi_pkt *);
143 static int sf_start_internal(struct sf *, struct sf_pkt *);
144 static void sf_fill_ids(struct sf *, struct sf_pkt *, struct sf_target *);
145 static int sf_prepare_pkt(struct sf *, struct sf_pkt *, struct sf_target *);
146 static int sf_dopoll(struct sf *, struct sf_pkt *);
147 static void sf_cmd_callback(struct fc_lun_packet *);
148 static void sf_throttle(struct sf *);
149 static void sf_watch(void *);
150 static void sf_throttle_start(struct sf *);
151 static void sf_check_targets(struct sf *);
152 static void sf_check_reset_delay(void *);
153 static int sf_target_timeout(struct sf *, struct sf_pkt *);
154 static void sf_force_lip(struct sf *);
155 static void sf_unsol_els_callback(void *, soc_response_t *, caddr_t);
156 static struct sf_els_hdr *sf_els_timeout(struct sf *, struct sf_els_hdr *);
157 /*PRINTFLIKE3*/
158 static void sf_log(struct sf *, int, const char *, ...);
159 static int sf_kstat_update(kstat_t *, int);
160 static int sf_open(dev_t *, int, int, cred_t *);
161 static int sf_close(dev_t *, int, int, cred_t *);
162 static int sf_ioctl(dev_t *, int, intptr_t, int, cred_t *, int *);
163 static struct sf_target *sf_get_target_from_dip(struct sf *, dev_info_t *);
164 static int sf_bus_get_eventcookie(dev_info_t *, dev_info_t *, char *,
165     ddi_eventcookie_t *);
166 static int sf_bus_add_eventcall(dev_info_t *, dev_info_t *,
167     ddi_eventcookie_t, void (*)(), void *, ddi_callback_id_t *cb_id);
168 static int sf_bus_remove_eventcall(dev_info_t *devi, ddi_callback_id_t cb_id);
169 static int sf_bus_post_event(dev_info_t *, dev_info_t *,
170     ddi_eventcookie_t, void *);

172 static void sf_hp_daemon(void *);

174 /*
175  * this is required to be able to supply a control node
176  * where ioctl's can be executed
177  */
178 struct cb_ops sf_cb_ops = {
179     sf_open,           /* open */
180     sf_close,        /* close */
181     nodev,           /* strategy */
182     nodev,           /* print */
183     nodev,           /* dump */
184     nodev,           /* read */
185     nodev,           /* write */
186     sf_ioctl,        /* ioctl */
187     nodev,           /* devmap */
188     nodev,           /* mmap */
189     nodev,           /* segmap */
190     nochpoll,       /* poll */

```

```

191     ddi_prop_op,     /* cb_prop_op */
192     0,               /* streamtab */
193     D_MP | D_NEW | D_HOTPLUG /* driver flags */
194 };
195 };

197 /*
198  * autoconfiguration routines.
199  */
200 static struct dev_ops sf_ops = {
201     DEVO_REV,       /* devo_rev, */
202     0,              /* refcnt */
203     sf_info,        /* info */
204     nulldev,       /* identify */
205     nulldev,       /* probe */
206     sf_attach,     /* attach */
207     sf_detach,     /* detach */
208     nodev,         /* reset */
209     &sf_cb_ops,     /* driver operations */
210     NULL,          /* bus operations */
211     NULL,          /* power management */
212     ddi_quiesce_not_supported, /* devo_quiesce */
213 };

215 #define SF_NAME "FC-AL FCP Nexus Driver" /* Name of the module. */
216 static char sf_version[] = "1.72 08/19/2008"; /* version of the module */

218 static struct modldrv modldrv = {
219     &mod_driverops, /* Type of module. This one is a driver */
220     SF_NAME,
221     &sf_ops,       /* driver ops */
222 };

224 static struct modlinkage modlinkage = {
225     MODREV_1, (void *)&modldrv, NULL
226 };

228 /* XXXXXXX The following is here to handle broken targets -- remove it later */
229 static int sf_reportlun_forever = 0;
230 /* XXXXXXX */
231 static int sf_lip_on_plogo = 0;
232 static int sf_els_retries = SF_ELS_RETRIES;
233 static struct sf *sf_head = NULL;
234 static int sf_target_scan_cnt = 4;
235 static int sf_pkt_scan_cnt = 5;
236 static int sf_pool_scan_cnt = 1800;
237 static void *sf_state = NULL;
238 static int sf_watchdog_init = 0;
239 static int sf_watchdog_time = 0;
240 static int sf_watchdog_timeout = 1;
241 static int sf_watchdog_tick;
242 static int sf_watch_running = 0;
243 static timeout_id_t sf_watchdog_id;
244 static timeout_id_t sf_reset_timeout_id;
245 static int sf_max_targets = SF_MAX_TARGETS;
246 static kmutex_t sf_global_mutex;
247 static int sf_core = 0;
248 int *sf_token = NULL; /* Must not be static or lint complains. */
249 static kcondvar_t sf_watch_cv;
250 extern pri_t minclsyspri;
251 static ddi_eventcookie_t sf_insert_eid;
252 static ddi_eventcookie_t sf_remove_eid;

254 static ndi_event_definition_t sf_event_defs[] = {
255     { SF_EVENT_TAG_INSERT, FCAL_INSERT_EVENT, EPL_KERNEL, 0 },
256     { SF_EVENT_TAG_REMOVE, FCAL_REMOVE_EVENT, EPL_INTERRUPT, 0 }

```

```

257 };
258
259 #define SF_N_NDI_EVENTS \
260     (sizeof (sf_event_defs) / sizeof (ndi_event_definition_t))
261
262 #ifdef DEBUG
263 static int sf_lip_flag = 1;          /* bool: to allow LIPs */
264 static int sf_reset_flag = 1;      /* bool: to allow reset after LIP */
265 static int sf_abort_flag = 0;     /* bool: to do just one abort */
266 #endif
267
268 extern int64_t ddi_get_lbolt64(void);
269
270 /*
271  * for converting between target number (switch) and hard address/AL_PA
272  */
273 static uchar_t sf_switch_to_alpa[] = {
274     0xef, 0xe8, 0xe4, 0xe2, 0xe1, 0xe0, 0xdc, 0xda, 0xd9, 0xd6,
275     0xd5, 0xd4, 0xd3, 0xd2, 0xd1, 0xce, 0xcd, 0xcc, 0xcb, 0xca,
276     0xc9, 0xc7, 0xc6, 0xc5, 0xc3, 0xbc, 0xba, 0xb9, 0xb6, 0xb5,
277     0xb4, 0xb3, 0xb2, 0xb1, 0xae, 0xad, 0xac, 0xab, 0xaa, 0xa9,
278     0xa7, 0xa6, 0xa5, 0xa3, 0x9f, 0x9e, 0x9d, 0x9b, 0x98, 0x97,
279     0x90, 0x8f, 0x88, 0x84, 0x82, 0x81, 0x80, 0x7c, 0x7a, 0x79,
280     0x76, 0x75, 0x74, 0x73, 0x72, 0x71, 0x6e, 0x6d, 0x6c, 0x6b,
281     0x6a, 0x69, 0x67, 0x66, 0x65, 0x63, 0x5c, 0x5a, 0x59, 0x56,
282     0x55, 0x54, 0x53, 0x52, 0x51, 0x4e, 0x4d, 0x4c, 0x4b, 0x4a,
283     0x49, 0x47, 0x46, 0x45, 0x43, 0x3c, 0x3a, 0x39, 0x36, 0x35,
284     0x34, 0x33, 0x32, 0x31, 0x2e, 0x2d, 0x2c, 0x2b, 0x2a, 0x29,
285     0x27, 0x26, 0x25, 0x23, 0x1f, 0x1e, 0x1d, 0x1b, 0x18, 0x17,
286     0x10, 0x0f, 0x08, 0x04, 0x02, 0x01
287 };
288
289 static uchar_t sf_alpa_to_switch[] = {
290     0x00, 0x7d, 0x7c, 0x00, 0x7b, 0x00, 0x00, 0x00, 0x7a, 0x00,
291     0x00, 0x00, 0x00, 0x00, 0x00, 0x79, 0x78, 0x00, 0x00, 0x00,
292     0x00, 0x00, 0x00, 0x77, 0x76, 0x00, 0x00, 0x75, 0x00, 0x74,
293     0x73, 0x72, 0x00, 0x00, 0x00, 0x71, 0x00, 0x70, 0x6f, 0x6e,
294     0x00, 0x6d, 0x6c, 0x6b, 0x6a, 0x69, 0x68, 0x00, 0x00, 0x67,
295     0x66, 0x65, 0x64, 0x63, 0x62, 0x00, 0x00, 0x61, 0x60, 0x00,
296     0x5f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x5e, 0x00, 0x5d,
297     0x5c, 0x5b, 0x00, 0x5a, 0x59, 0x58, 0x57, 0x56, 0x55, 0x00,
298     0x00, 0x54, 0x53, 0x52, 0x51, 0x50, 0x4f, 0x00, 0x00, 0x4e,
299     0x4d, 0x00, 0x4c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x4b,
300     0x00, 0x4a, 0x49, 0x48, 0x00, 0x47, 0x46, 0x45, 0x44, 0x43,
301     0x42, 0x00, 0x00, 0x41, 0x40, 0x3f, 0x3e, 0x3d, 0x3c, 0x00,
302     0x00, 0x3b, 0x3a, 0x00, 0x39, 0x00, 0x00, 0x00, 0x38, 0x37,
303     0x36, 0x00, 0x35, 0x00, 0x00, 0x34, 0x00, 0x00, 0x00, 0x00,
304     0x00, 0x00, 0x00, 0x33, 0x32, 0x00, 0x00, 0x00, 0x00, 0x00,
305     0x00, 0x31, 0x30, 0x00, 0x00, 0x2f, 0x00, 0x2e, 0x2d, 0x2c,
306     0x00, 0x00, 0x00, 0x2b, 0x00, 0x2a, 0x29, 0x28, 0x00, 0x27,
307     0x26, 0x25, 0x24, 0x23, 0x22, 0x00, 0x00, 0x21, 0x20, 0x1f,
308     0x1e, 0x1d, 0x1c, 0x00, 0x00, 0x00, 0x1b, 0x1a, 0x00, 0x19, 0x00,
309     0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x00, 0x17, 0x16, 0x15,
310     0x00, 0x14, 0x13, 0x12, 0x11, 0x10, 0x0f, 0x00, 0x00, 0x0e,
311     0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x00, 0x00, 0x08, 0x07, 0x00,
312     0x06, 0x00, 0x00, 0x00, 0x05, 0x04, 0x03, 0x00, 0x02, 0x00,
313     0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
314 };
315
316 /*
317  * these macros call the proper transport-layer function given
318  * a particular transport
319  */
320 #define soc_transport(a, b, c, d) (*a->fcal_ops->fcal_transport)(b, c, d)
321 #define soc_transport_poll(a, b, c, d)\
322     (*a->fcal_ops->fcal_transport_poll)(b, c, d)

```

```

323 #define soc_get_lilp_map(a, b, c, d, e)\
324     (*a->fcal_ops->fcal_lilp_map)(b, c, d, e)
325 #define soc_force_lip(a, b, c, d, e)\
326     (*a->fcal_ops->fcal_force_lip)(b, c, d, e)
327 #define soc_abort(a, b, c, d, e)\
328     (*a->fcal_ops->fcal_abort_cmd)(b, c, d, e)
329 #define soc_force_reset(a, b, c, d)\
330     (*a->fcal_ops->fcal_force_reset)(b, c, d)
331 #define soc_add_ulp(a, b, c, d, e, f, g, h)\
332     (*a->fcal_ops->fcal_add_ulp)(b, c, d, e, f, g, h)
333 #define soc_remove_ulp(a, b, c, d, e)\
334     (*a->fcal_ops->fcal_remove_ulp)(b, c, d, e)
335 #define soc_take_core(a, b) (*a->fcal_ops->fcal_take_core)(b)
336
337
338 /* power management property defines (should be in a common include file?) */
339 #define PM_HARDWARE_STATE_PROP          "pm-hardware-state"
340 #define PM_NEEDS_SUSPEND_RESUME        "needs-suspend-resume"
341
342
343 /* node properties */
344 #define NODE_WWN_PROP                   "node-wwn"
345 #define PORT_WWN_PROP                   "port-wwn"
346 #define LIP_CNT_PROP                    "lip-count"
347 #define TARGET_PROP                     "target"
348 #define LUN_PROP                         "lun"
349
350
351 /*
352  * initialize this driver and install this module
353  */
354 int
355 _init(void)
356 {
357     int    i;
358
359     i = ddi_soft_state_init(&sf_state, sizeof (struct sf),
360         SF_INIT_ITEMS);
361     if (i != 0)
362         return (i);
363
364     if ((i = scsi_hba_init(&modlinkage)) != 0) {
365         ddi_soft_state_fini(&sf_state);
366         return (i);
367     }
368
369     mutex_init(&sf_global_mutex, NULL, MUTEX_DRIVER, NULL);
370     sf_watch_running = 0;
371     cv_init(&sf_watch_cv, NULL, CV_DRIVER, NULL);
372
373     if ((i = mod_install(&modlinkage)) != 0) {
374         mutex_destroy(&sf_global_mutex);
375         cv_destroy(&sf_watch_cv);
376         scsi_hba_fini(&modlinkage);
377         ddi_soft_state_fini(&sf_state);
378         return (i);
379     }
380
381     return (i);
382 }
383
384 /*
385  * remove this driver module from the system
386  */
387 int
388 _fini(void)

```

```

389 _fini(void)
390 {
391     int    i;

393     if ((i = mod_remove(&modlinkage)) == 0) {
394         scsi_hba_fini(&modlinkage);
395         mutex_destroy(&sf_global_mutex);
396         cv_destroy(&sf_watch_cv);
397         ddi_soft_state_fini(&sf_state);
398     }
399     return (i);
400 }

403 int
404 _info(struct modinfo *modinfop)
405 {
406     return (mod_info(&modlinkage, modinfop));
407 }

409 /*
410  * Given the device number return the devinfo pointer or instance
411  */
412 /*ARGSUSED*/
413 static int
414 sf_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
415 {
416     int          instance = SF_MINOR2INST(getminor((dev_t)arg));
417     struct sf    *sf;

419     switch (infocmd) {
420     case DDI_INFO_DEVT2DEVINFO:
421         sf = ddi_get_soft_state(sf_state, instance);
422         if (sf != NULL)
423             *result = sf->sf_dip;
424         else {
425             *result = NULL;
426             return (DDI_FAILURE);
427         }
428         break;

430     case DDI_INFO_DEVT2INSTANCE:
431         *result = (void *) (uintptr_t) instance;
432         break;
433     default:
434         return (DDI_FAILURE);
435     }
436     return (DDI_SUCCESS);
437 }

439 /*
440  * either attach or resume this driver
441  */
442 static int
443 sf_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
444 {
445     int instance;
446     int mutex_inittd = FALSE;
447     uint_t ccount;
448     size_t i, real_size;
449     struct fcsl_transport *handle;
450     char buf[64];
451     struct sf *sf, *tsf;
452     scsi_hba_tran_t *tran = NULL;
453     int handle_bound = FALSE;
454     kthread_t *tp;

```

```

457     switch ((int)cmd) {
459     case DDI_RESUME:

461         /*
462          * we've previously been SF_STATE_OFFLINED by a DDI_SUSPEND,
463          * so time to undo that and get going again by forcing a
464          * lip
465          */

467         instance = ddi_get_instance(dip);

469         sf = ddi_get_soft_state(sf_state, instance);
470         SF_DEBUG(2, (sf, CE_CONT,
471         "sf_attach: DDI_RESUME for sf%d\n", instance));
472         if (sf == NULL) {
473             cmn_err(CE_WARN, "sf%d: bad soft state", instance);
474             return (DDI_FAILURE);
475         }

477         /*
478          * clear suspended flag so that normal operations can resume
479          */
480         mutex_enter(&sf->sf_mutex);
481         sf->sf_state &= ~SF_STATE_SUSPENDED;
482         mutex_exit(&sf->sf_mutex);

484         /*
485          * force a login by setting our state to offline
486          */
487         sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
488         sf->sf_state = SF_STATE_OFFLINE;

490         /*
491          * call transport routine to register state change and
492          * ELS callback routines (to register us as a ULP)
493          */
494         soc_add_ulp(sf->sf_sochandle, sf->sf_socp,
495         sf->sf_sochandle->fcsl_portno, TYPE_SCSI_FCP,
496         sf_statecb_callback, sf_unsol_els_callback, NULL, sf);

498         /*
499          * call transport routine to force loop initialization
500          */
501         (void) soc_force_lip(sf->sf_sochandle, sf->sf_socp,
502         sf->sf_sochandle->fcsl_portno, 0, FCAL_NO_LIP);

504         /*
505          * increment watchdog init flag, setting watchdog timeout
506          * if we are the first (since somebody has to do it)
507          */
508         mutex_enter(&sf_global_mutex);
509         if (!sf_watchdog_init++) {
510             mutex_exit(&sf_global_mutex);
511             sf_watchdog_id = timeout(sf_watch,
512             (caddr_t)0, sf_watchdog_tick);
513         } else {
514             mutex_exit(&sf_global_mutex);
515         }

517         return (DDI_SUCCESS);

519     case DDI_ATTACH:

```

```

521      /*
522      * this instance attaching for the first time
523      */
524
525      instance = ddi_get_instance(dip);
526
527      if (ddi_soft_state_zalloc(sf_state, instance) !=
528          DDI_SUCCESS) {
529          cmn_err(CE_WARN, "sf%d: failed to allocate soft state",
530              instance);
531          return (DDI_FAILURE);
532      }
533
534      sf = ddi_get_soft_state(sf_state, instance);
535      SF_DEBUG(4, (sf, CE_CONT,
536          "sf_attach: DDI_ATTACH for sf%d\n", instance));
537      if (sf == NULL) {
538          /* this shouldn't happen since we just allocated it */
539          cmn_err(CE_WARN, "sf%d: bad soft state", instance);
540          return (DDI_FAILURE);
541      }
542
543      /*
544      * from this point on, if there's an error, we must de-allocate
545      * soft state before returning DDI_FAILURE
546      */
547
548      if ((handle = ddi_get_parent_data(dip)) == NULL) {
549          cmn_err(CE_WARN,
550              "sf%d: failed to obtain transport handle",
551              instance);
552          goto fail;
553      }
554
555      /* fill in our soft state structure */
556      sf->sf_dip = dip;
557      sf->sf_state = SF_STATE_INIT;
558      sf->sf_throttle = handle->fcsl_cmdmax;
559      sf->sf_sochandle = handle;
560      sf->sf_socp = handle->fcsl_handle;
561      sf->sf_check_n_close = 0;
562
563      /* create a command/response buffer pool for this instance */
564      if (sf_add_cr_pool(sf) != DDI_SUCCESS) {
565          cmn_err(CE_WARN,
566              "sf%d: failed to allocate command/response pool",
567              instance);
568          goto fail;
569      }
570
571      /* create a cache for this instance */
572      (void) sprintf(buf, "sf%d_cache", instance);
573      sf->sf_pkt_cache = kmem_cache_create(buf,
574          sizeof (fcsl_packet_t) + sizeof (struct sf_pkt) +
575          scsi_pkt_size(), 8,
576          sf_kmem_cache_constructor, sf_kmem_cache_destructor,
577          NULL, NULL, 0);
578      if (sf->sf_pkt_cache == NULL) {
579          cmn_err(CE_WARN, "sf%d: failed to allocate kmem cache",
580              instance);
581          goto fail;
582      }
583
584      /* set up a handle and allocate memory for DMA */
585      if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->
586          fcsl_dmaattr, DDI_DMA_DONTWAIT, NULL, &sf->

```

```

587          sf_lilp_dmahandle) != DDI_SUCCESS) {
588          cmn_err(CE_WARN,
589              "sf%d: failed to allocate dma handle for lilp map",
590              instance);
591          goto fail;
592      }
593      i = sizeof (struct fcsl_lilp_map) + 1;
594      if (ddi_dma_mem_alloc(sf->sf_lilp_dmahandle,
595          i, sf->sf_sochandle->
596          fcsl_accattr, DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT, NULL,
597          (caddr_t *)&sf->sf_lilp_map, &real_size,
598          &sf->sf_lilp_acchandle) != DDI_SUCCESS) {
599          cmn_err(CE_WARN, "sf%d: failed to allocate lilp map",
600              instance);
601          goto fail;
602      }
603      if (real_size < i) {
604          /* no error message ??? */
605          goto fail; /* trouble allocating memory */
606      }
607
608      /*
609      * set up the address for the DMA transfers (getting a cookie)
610      */
611      if (ddi_dma_addr_bind_handle(sf->sf_lilp_dmahandle, NULL,
612          (caddr_t)sf->sf_lilp_map, real_size,
613          DDI_DMA_READ | DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT, NULL,
614          &sf->sf_lilp_dmacookie, &ccount) != DDI_DMA_MAPPED) {
615          cmn_err(CE_WARN,
616              "sf%d: failed to bind dma handle for lilp map",
617              instance);
618          goto fail;
619      }
620      handle_bound = TRUE;
621      /* ensure only one cookie was allocated */
622      if (ccount != 1) {
623          goto fail;
624      }
625
626      /* ensure LILP map and DMA cookie addresses are even?? */
627      sf->sf_lilp_map = (struct fcsl_lilp_map *)(((uintptr_t)sf->
628          sf_lilp_map + 1) & ~1);
629      sf->sf_lilp_dmacookie.dmac_address = (sf->
630          sf_lilp_dmacookie.dmac_address + 1) & ~1;
631
632      /* set up all of our mutexes and condition variables */
633      mutex_init(&sf->sf_mutex, NULL, MUTEX_DRIVER, NULL);
634      mutex_init(&sf->sf_cmd_mutex, NULL, MUTEX_DRIVER, NULL);
635      mutex_init(&sf->sf_cr_mutex, NULL, MUTEX_DRIVER, NULL);
636      mutex_init(&sf->sf_hp_daemon_mutex, NULL, MUTEX_DRIVER, NULL);
637      cv_init(&sf->sf_cr_cv, NULL, CV_DRIVER, NULL);
638      cv_init(&sf->sf_hp_daemon_cv, NULL, CV_DRIVER, NULL);
639
640      mutex_init(&sf->sf_mutex, NULL, MUTEX_DRIVER, NULL);
641
642      /* create our devctl minor node */
643      if (ddi_create_minor_node(dip, "devctl", S_IFCHR,
644          SF_INST2DEVCTL_MINOR(instance),
645          DDI_NT_NEXUS, 0) != DDI_SUCCESS) {
646          cmn_err(CE_WARN, "sf%d: ddi_create_minor_node failed"
647              " for devctl", instance);
648          goto fail;
649      }
650
651      /* create fc minor node */
652      if (ddi_create_minor_node(dip, "fc", S_IFCHR,

```



```

653         SF_INST2FC_MINOR(instance), DDI_NT_FC_ATTACHMENT_POINT,
654         0) != DDI_SUCCESS) {
655             cmn_err(CE_WARN, "sf%d: ddi_create_minor_node failed"
656                  " for fc", instance);
657             goto fail;
658         }
659         /* allocate a SCSI transport structure */
660         tran = scsi_hba_tran_alloc(dip, 0);
661         if (tran == NULL) {
662             /* remove all minor nodes created */
663             ddi_remove_minor_node(dip, NULL);
664             cmn_err(CE_WARN, "sf%d: scsi_hba_tran_alloc failed",
665                  instance);
666             goto fail;
667         }
669         /* Indicate that we are 'sizeof (scsi_*(9S))' clean. */
670         scsi_size_clean(dip);          /* SCSI_SIZE_CLEAN_VERIFY ok */
672         /* save ptr to new transport structure and fill it in */
673         sf->sf_tran = tran;
675         tran->tran_hba_private         = sf;
676         tran->tran_tgt_private         = NULL;
677         tran->tran_tgt_init            = sf_scsi_tgt_init;
678         tran->tran_tgt_probe          = NULL;
679         tran->tran_tgt_free            = sf_scsi_tgt_free;
681         tran->tran_start                = sf_start;
682         tran->tran_abort                = sf_abort;
683         tran->tran_reset                = sf_reset;
684         tran->tran_getcap               = sf_getcap;
685         tran->tran_setcap               = sf_setcap;
686         tran->tran_init_pkt             = sf_scsi_init_pkt;
687         tran->tran_destroy_pkt          = sf_scsi_destroy_pkt;
688         tran->tran_dmafree              = sf_scsi_dmafree;
689         tran->tran_sync_pkt             = sf_scsi_sync_pkt;
690         tran->tran_reset_notify         = sf_scsi_reset_notify;
692         /*
693          * register event notification routines with scsa
694          */
695         tran->tran_get_eventcookie      = sf_bus_get_eventcookie;
696         tran->tran_add_eventcall        = sf_bus_add_eventcall;
697         tran->tran_remove_eventcall     = sf_bus_remove_eventcall;
698         tran->tran_post_event           = sf_bus_post_event;
700         /*
701          * register bus configure/unconfigure
702          */
703         tran->tran_bus_config           = sf_scsi_bus_config;
704         tran->tran_bus_unconfig         = sf_scsi_bus_unconfig;
706         /*
707          * allocate an ndi event handle
708          */
709         sf->sf_event_defs = (ndi_event_definition_t *)
710             kmem_zalloc(sizeof (sf_event_defs), KM_SLEEP);
712         bcopy(sf_event_defs, sf->sf_event_defs,
713             sizeof (sf_event_defs));
715         (void) ndi_event_alloc_hdl(dip, NULL,
716             &sf->sf_event_hdl, NDI_SLEEP);
718         sf->sf_events.ndi_events_version = NDI_EVENTS_REV1;

```

```

719         sf->sf_events.ndi_n_events = SF_N_NDI_EVENTS;
720         sf->sf_events.ndi_event_defs = sf->sf_event_defs;
722         if (ndi_event_bind_set(sf->sf_event_hdl,
723             &sf->sf_events, NDI_SLEEP) != NDI_SUCCESS) {
724             goto fail;
725         }
727         tran->tran_get_name             = sf_scsi_get_name;
728         tran->tran_get_bus_addr         = sf_scsi_get_bus_addr;
730         /* setup and attach SCSI hba transport */
731         if (scsi_hba_attach_setup(dip, sf->sf_sochandle->
732             fcal_dmaattr, tran, SCSI_HBA_TRAN_CLONE) != DDI_SUCCESS) {
733             cmn_err(CE_WARN, "sf%d: scsi_hba_attach_setup failed",
734                  instance);
735             goto fail;
736         }
738         /* set up kstats */
739         if ((sf->sf_ksp = kstat_create("sf", instance, "statistics",
740             "controller", KSTAT_TYPE_RAW, sizeof (struct sf_stats),
741             KSTAT_FLAG_VIRTUAL)) == NULL) {
742             cmn_err(CE_WARN, "sf%d: failed to create kstat",
743                  instance);
744         } else {
745             sf->sf_stats.version = 2;
746             (void) sprintf(sf->sf_stats.drivr_name,
747                 "%s: %s", SF_NAME, sf_version);
748             sf->sf_ksp->ks_data = (void *)&sf->sf_stats;
749             sf->sf_ksp->ks_private = sf;
750             sf->sf_ksp->ks_update = sf_kstat_update;
751             kstat_install(sf->sf_ksp);
752         }
754         /* create the hotplug thread */
755         mutex_enter(&sf->sf_hp_daemon_mutex);
756         tp = thread_create(NULL, 0,
757             (void (*)(void))sf_hp_daemon, sf, 0, &p0, TS_RUN, minclsyspri);
758         sf->sf_hp_tid = tp->t_did;
759         mutex_exit(&sf->sf_hp_daemon_mutex);
761         /* add this soft state instance to the head of the list */
762         mutex_enter(&sf_global_mutex);
763         sf->sf_next = sf_head;
764         tsf = sf_head;
765         sf_head = sf;
767         /*
768          * find entry in list that has the same FC-AL handle (if any)
769          */
770         while (tsf != NULL) {
771             if (tsf->sf_socp == sf->sf_socp) {
772                 break;          /* found matching entry */
773             }
774             tsf = tsf->sf_next;
775         }
777         if (tsf != NULL) {
778             /* if we found a matching entry keep track of it */
779             sf->sf_sibling = tsf;
780         }
782         /*
783          * increment watchdog init flag, setting watchdog timeout
784          * if we are the first (since somebody has to do it)

```

```

785     */
786     if (!sf_watchdog_init++) {
787         mutex_exit(&sf_global_mutex);
788         sf_watchdog_tick = sf_watchdog_timeout *
789             drv_usectohz(1000000);
790         sf_watchdog_id = timeout(sf_watch,
791             NULL, sf_watchdog_tick);
792     } else {
793         mutex_exit(&sf_global_mutex);
794     }
795
796     if (tsf != NULL) {
797         /*
798          * set up matching entry to be our sibling
799          */
800         mutex_enter(&tsf->sf_mutex);
801         tsf->sf_sibling = sf;
802         mutex_exit(&tsf->sf_mutex);
803     }
804
805     /*
806     * create this property so that PM code knows we want
807     * to be suspended at PM time
808     */
809     (void) ddi_prop_update_string(DDI_DEV_T_NONE, dip,
810         PM_HARDWARE_STATE_PROP, PM_NEEDS_SUSPEND_RESUME);
811
812     /* log the fact that we have a new device */
813     ddi_report_dev(dip);
814
815     /*
816     * force a login by setting our state to offline
817     */
818     sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
819     sf->sf_state = SF_STATE_OFFLINE;
820
821     /*
822     * call transport routine to register state change and
823     * ELS callback routines (to register us as a ULP)
824     */
825     soc_add_ulp(sf->sf_sochandle, sf->sf_socp,
826         sf->sf_sochandle->fcml_portno, TYPE_SCSI_FCP,
827         sf_statecb_callback, sf_unsol_els_callback, NULL, sf);
828
829     /*
830     * call transport routine to force loop initialization
831     */
832     (void) soc_force_lip(sf->sf_sochandle, sf->sf_socp,
833         sf->sf_sochandle->fcml_portno, 0, FCAL_NO_LIP);
834     sf->sf_reset_time = ddi_get_lbolt64();
835     return (DDI_SUCCESS);
836
837 default:
838     return (DDI_FAILURE);
839 }
840
841 fail:
842     cmn_err(CE_WARN, "sf%d: failed to attach", instance);
843
844     /*
845     * Unbind and free event set
846     */
847     if (sf->sf_event_hdl) {
848         (void) ndi_event_unbind_set(sf->sf_event_hdl,
849             &sf->sf_events, NDI_SLEEP);
850         (void) ndi_event_free_hdl(sf->sf_event_hdl);

```

```

851     }
852
853     if (sf->sf_event_defs) {
854         kmem_free(sf->sf_event_defs, sizeof (sf_event_defs));
855     }
856
857     if (sf->sf_tran != NULL) {
858         scsi_hba_tran_free(sf->sf_tran);
859     }
860     while (sf->sf_cr_pool != NULL) {
861         sf_crpool_free(sf);
862     }
863     if (sf->sf_lilp_dmahandle != NULL) {
864         if (handle_bound) {
865             (void) ddi_dma_unbind_handle(sf->sf_lilp_dmahandle);
866         }
867         ddi_dma_free_handle(&sf->sf_lilp_dmahandle);
868     }
869     if (sf->sf_pkt_cache != NULL) {
870         kmem_cache_destroy(sf->sf_pkt_cache);
871     }
872     if (sf->sf_lilp_map != NULL) {
873         ddi_dma_mem_free(&sf->sf_lilp_acchandle);
874     }
875     if (sf->sf_ksp != NULL) {
876         kstat_delete(sf->sf_ksp);
877     }
878     if (mutex_init) {
879         mutex_destroy(&sf->sf_mutex);
880         mutex_destroy(&sf->sf_cmd_mutex);
881         mutex_destroy(&sf->sf_cr_mutex);
882         mutex_destroy(&sf->sf_hp_daemon_mutex);
883         cv_destroy(&sf->sf_cr_cv);
884         cv_destroy(&sf->sf_hp_daemon_cv);
885     }
886     mutex_enter(&sf_global_mutex);
887
888     /*
889     * kill off the watchdog if we are the last instance
890     */
891     if (!--sf_watchdog_init) {
892         timeout_id_t tid = sf_watchdog_id;
893         mutex_exit(&sf_global_mutex);
894         (void) untimedout(tid);
895     } else {
896         mutex_exit(&sf_global_mutex);
897     }
898
899     ddi_soft_state_free(sf_state, instance);
900
901     if (tran != NULL) {
902         /* remove all minor nodes */
903         ddi_remove_minor_node(dip, NULL);
904     }
905
906     return (DDI_FAILURE);
907 }
908
909 /* ARGSUSED */
910 static int
911 sf_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
912 {
913     struct sf *sf;
914     int instance;
915     int i;

```

```

917 struct sf_target *target;
918 timeout_id_t tid;

922 /* NO OTHER THREADS ARE RUNNING */

924 instance = ddi_get_instance(dip);

926 if ((sf = ddi_get_soft_state(sf_state, instance)) == NULL) {
927     cmn_err(CE_WARN, "sf_detach, sf%d: bad soft state", instance);
928     return (DDI_FAILURE);
929 }

931 switch (cmd) {

933 case DDI_SUSPEND:
934     /*
935      * suspend our instance
936      */

938     SF_DEBUG(2, (sf, CE_CONT,
939                 "sf_detach: DDI_SUSPEND for sf%d\n", instance));
940     /*
941      * There is a race condition in socall where while doing
942      * callbacks if a ULP removes it self from the callback list
943      * the for loop in socall may panic as cblst is junk and
944      * while trying to get cblst->next the system will panic.
945      */

947     /* call transport to remove our unregisters our callbacks */
948     soc_remove_ulp(sf->sf_sochandle, sf->sf_socp,
949                  sf->sf_sochandle->fcall_portno, TYPE_SCSI_FCP, sf);

951     /*
952      * begin process of clearing outstanding commands
953      * by issuing a lip
954      */
955     sf_force_lip(sf);

957     /*
958      * toggle the device OFFLINE in order to cause
959      * outstanding commands to drain
960      */
961     mutex_enter(&sf->sf_mutex);
962     sf->sf_lip_cnt++;
963     sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
964     sf->sf_state = (SF_STATE_OFFLINE | SF_STATE_SUSPENDED);
965     for (i = 0; i < sf_max_targets; i++) {
966         target = sf->sf_targets[i];
967         if (target != NULL) {
968             struct sf_target *ntarget;

970             mutex_enter(&target->sft_mutex);
971             if (!(target->sft_state & SF_TARGET_OFFLINE)) {
972                 target->sft_state |=
973                     (SF_TARGET_BUSY | SF_TARGET_MARK);
974             }
975             /* do this for all LUNs as well */
976             for (ntarget = target->sft_next_lun;
977                  ntarget;
978                  ntarget = ntarget->sft_next_lun) {
979                 mutex_enter(&ntarget->sft_mutex);
980                 if (!(ntarget->sft_state &
981                     SF_TARGET_OFFLINE)) {
982                     ntarget->sft_state |=

```

```

983                                     (SF_TARGET_BUSY |
984                                     SF_TARGET_MARK);
985                                     }
986                                     mutex_exit(&target->sft_mutex);
987                                     }
988                                     mutex_exit(&target->sft_mutex);
989                                     }
990     }
991     mutex_exit(&sf->sf_mutex);
992     mutex_enter(&sf_global_mutex);

994     /*
995      * kill off the watchdog if we are the last instance
996      */
997     if (!--sf_watchdog_init) {
998         tid = sf_watchdog_id;
999         mutex_exit(&sf_global_mutex);
1000         (void) untimeout(tid);
1001     } else {
1002         mutex_exit(&sf_global_mutex);
1003     }

1005     return (DDI_SUCCESS);

1007 case DDI_DETACH:
1008     /*
1009      * detach this instance
1010      */

1012     SF_DEBUG(2, (sf, CE_CONT,
1013                 "sf_detach: DDI_DETACH for sf%d\n", instance));

1015     /* remove this "sf" from the list of sf softstates */
1016     sf_softstate_unlink(sf);

1018     /*
1019      * prior to taking any DDI_DETACH actions, toggle the
1020      * device OFFLINE in order to cause outstanding
1021      * commands to drain
1022      */
1023     mutex_enter(&sf->sf_mutex);
1024     sf->sf_lip_cnt++;
1025     sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
1026     sf->sf_state = SF_STATE_OFFLINE;
1027     for (i = 0; i < sf_max_targets; i++) {
1028         target = sf->sf_targets[i];
1029         if (target != NULL) {
1030             struct sf_target *ntarget;

1032             mutex_enter(&target->sft_mutex);
1033             if (!(target->sft_state & SF_TARGET_OFFLINE)) {
1034                 target->sft_state |=
1035                     (SF_TARGET_BUSY | SF_TARGET_MARK);
1036             }
1037             for (ntarget = target->sft_next_lun;
1038                  ntarget;
1039                  ntarget = ntarget->sft_next_lun) {
1040                 mutex_enter(&ntarget->sft_mutex);
1041                 if (!(ntarget->sft_state &
1042                     SF_TARGET_OFFLINE)) {
1043                     ntarget->sft_state |=
1044                         (SF_TARGET_BUSY |
1045                         SF_TARGET_MARK);
1046                 }
1047                 mutex_exit(&ntarget->sft_mutex);
1048             }

```

```

1049         mutex_exit(&target->sft_mutex);
1050     }
1051 }
1052 mutex_exit(&sf->sf_mutex);

1054 /* call transport to remove and unregister our callbacks */
1055 soc_remove_ulp(sf->sf_sochandle, sf->sf_socp,
1056             sf->sf_sochandle->fcal_portno, TYPE_SCSI_FCP, sf);

1058 /*
1059  * kill off the watchdog if we are the last instance
1060  */
1061 mutex_enter(&sf_global_mutex);
1062 if (!--sf_watchdog_init) {
1063     tid = sf_watchdog_id;
1064     mutex_exit(&sf_global_mutex);
1065     (void) untimeout(tid);
1066 } else {
1067     mutex_exit(&sf_global_mutex);
1068 }

1070 /* signal sf_hp_daemon() to exit and wait for exit */
1071 mutex_enter(&sf->sf_hp_daemon_mutex);
1072 ASSERT(sf->sf_hp_tid);
1073 sf->sf_hp_exit = 1; /* flag exit */
1074 cv_signal(&sf->sf_hp_daemon_cv);
1075 mutex_exit(&sf->sf_hp_daemon_mutex);
1076 thread_join(sf->sf_hp_tid); /* wait for hotplug to exit */

1078 /*
1079  * Unbind and free event set
1080  */
1081 if (sf->sf_event_hdl) {
1082     (void) ndi_event_unbind_set(sf->sf_event_hdl,
1083         &sf->sf_events, NDI_SLEEP);
1084     (void) ndi_event_free_hdl(sf->sf_event_hdl);
1085 }

1087 if (sf->sf_event_defs) {
1088     kmem_free(sf->sf_event_defs, sizeof (sf_event_defs));
1089 }

1091 /* detach this instance of the HBA driver */
1092 (void) scsi_hba_detach(dip);
1093 scsi_hba_tran_free(sf->sf_tran);

1095 /* deallocate/unbind DMA handle for lilp map */
1096 if (sf->sf_lilp_map != NULL) {
1097     (void) ddi_dma_unbind_handle(sf->sf_lilp_dmahandle);
1098     if (sf->sf_lilp_dmahandle != NULL) {
1099         ddi_dma_free_handle(&sf->sf_lilp_dmahandle);
1100     }
1101     ddi_dma_mem_free(&sf->sf_lilp_acchandle);
1102 }

1104 /*
1105  * the kmem cache must be destroyed before free'ing
1106  * up the crpools
1107  *
1108  * our finagle of "ntot" and "nfree"
1109  * causes an ASSERT failure in "sf_cr_free()"
1110  * if the kmem cache is free'd after invoking
1111  * "sf_crpool_free()".
1112  */
1113 kmem_cache_destroy(sf->sf_pkt_cache);

```

```

1115     SF_DEBUG(2, (sf, CE_CONT,
1116         "sf_detach: sf_crpool_free() for instance 0x%x\n",
1117         instance));
1118     while (sf->sf_cr_pool != NULL) {
1119         /*
1120          * set ntot to nfree for this particular entry
1121          *
1122          * this causes sf_crpool_free() to update
1123          * the cr_pool list when deallocating this entry
1124          */
1125         sf->sf_cr_pool->ntot = sf->sf_cr_pool->nfree;
1126         sf_crpool_free(sf);
1127     }

1129     /*
1130     * now that the cr_pool's are gone it's safe
1131     * to destroy all softstate mutex's and cv's
1132     */
1133     mutex_destroy(&sf->sf_mutex);
1134     mutex_destroy(&sf->sf_cmd_mutex);
1135     mutex_destroy(&sf->sf_cr_mutex);
1136     mutex_destroy(&sf->sf_hp_daemon_mutex);
1137     cv_destroy(&sf->sf_cr_cv);
1138     cv_destroy(&sf->sf_hp_daemon_cv);

1140     /* remove all minor nodes from the device tree */
1141     ddi_remove_minor_node(dip, NULL);

1143     /* remove properties created during attach() */
1144     ddi_prop_remove_all(dip);

1146     /* remove kstat's if present */
1147     if (sf->sf_ksp != NULL) {
1148         kstat_delete(sf->sf_ksp);
1149     }

1151     SF_DEBUG(2, (sf, CE_CONT,
1152         "sf_detach: ddi_soft_state_free() for instance 0x%x\n",
1153         instance));
1154     ddi_soft_state_free(sf_state, instance);
1155     return (DDI_SUCCESS);

1157     default:
1158         SF_DEBUG(2, (sf, CE_CONT, "sf_detach: sf%d unknown cmd %x\n",
1159             instance, (int)cmd));
1160         return (DDI_FAILURE);
1161     }
1162 }

1165 /*
1166  * sf_softstate_unlink() - remove an sf instance from the list of softstates
1167  */
1168 static void
1169 sf_softstate_unlink(struct sf *sf)
1170 {
1171     struct sf *sf_ptr;
1172     struct sf *sf_found_sibling;
1173     struct sf *sf_reposition = NULL;

1176     mutex_enter(&sf_global_mutex);
1177     while (sf_watch_running) {
1178         /* Busy working the list -- wait */
1179         cv_wait(&sf_watch_cv, &sf_global_mutex);
1180     }

```

```

1181     if ((sf_found_sibling = sf->sf_sibling) != NULL) {
1182         /*
1183          * we have a sibling so NULL out its reference to us
1184          */
1185         mutex_enter(&sf_found_sibling->sf_mutex);
1186         sf_found_sibling->sf_sibling = NULL;
1187         mutex_exit(&sf_found_sibling->sf_mutex);
1188     }

1190     /* remove our instance from the global list */
1191     if (sf == sf_head) {
1192         /* we were at at head of the list */
1193         sf_head = sf->sf_next;
1194     } else {
1195         /* find us in the list */
1196         for (sf_ptr = sf_head;
1197             sf_ptr != NULL;
1198             sf_ptr = sf_ptr->sf_next) {
1199             if (sf_ptr == sf) {
1200                 break;
1201             }
1202             /* remember this place */
1203             sf_reposition = sf_ptr;
1204         }
1205         ASSERT(sf_ptr == sf);
1206         ASSERT(sf_reposition != NULL);

1208         sf_reposition->sf_next = sf_ptr->sf_next;
1209     }
1210     mutex_exit(&sf_global_mutex);
1211 }

1214 static int
1215 sf_scsi_bus_config(dev_info_t *parent, uint_t flag,
1216                   ddi_bus_config_op_t op, void *arg, dev_info_t **childp)
1217 {
1218     int64_t      reset_delay;
1219     struct sf    *sf;

1221     sf = ddi_get_soft_state(sf_state, ddi_get_instance(parent));
1222     ASSERT(sf);

1224     reset_delay = (int64_t)(USEC_TO_TICK(SF_INIT_WAIT_TIMEOUT)) -
1225                   (ddi_get_lbolt64() - sf->sf_reset_time);
1226     if (reset_delay < 0)
1227         reset_delay = 0;

1229     if (sf_bus_config_debug)
1230         flag |= NDI_DEVI_DEBUG;

1232     return (ndi_busop_bus_config(parent, flag, op,
1233                                  arg, childp, (clock_t)reset_delay));
1234 }

1236 static int
1237 sf_scsi_bus_unconfig(dev_info_t *parent, uint_t flag,
1238                     ddi_bus_config_op_t op, void *arg)
1239 {
1240     if (sf_bus_config_debug)
1241         flag |= NDI_DEVI_DEBUG;

1243     return (ndi_busop_bus_unconfig(parent, flag, op, arg));
1244 }

```

```

1247 /*
1248  * called by transport to initialize a SCSI target
1249  */
1250 /* ARGSUSED */
1251 static int
1252 sf_scsi_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
1253                  scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
1254 {
1255     #ifdef RAID_LUNS
1256         int lun;
1257     #else
1258         int64_t lun;
1259     #endif
1260     struct sf_target *target;
1261     struct sf *sf = (struct sf *)hba_tran->tran_hba_private;
1262     int i, t_len;
1263     unsigned int lip_cnt;
1264     unsigned char wwn[FC_WWN_SIZE];

1267     /* get and validate our SCSI target ID */
1268     i = sd->sd_address.a_target;
1269     if (i >= sf_max_targets) {
1270         return (DDI_NOT_WELL_FORMED);
1271     }

1273     /* get our port WWN property */
1274     t_len = sizeof (wwn);
1275     if (ddi_prop_op(DDI_DEV_T_ANY, tgt_dip, PROP_LEN_AND_VAL_BUF,
1276                   DDI_PROP_DONTPASS | DDI_PROP_CANSLEEP, PORT_WWN_PROP,
1277                   (caddr_t)&wwn, &t_len) != DDI_SUCCESS) {
1278         /* no port WWN property - ignore the OBP stub node */
1279         return (DDI_NOT_WELL_FORMED);
1280     }

1282     /* get our LIP count property */
1283     t_len = sizeof (lip_cnt);
1284     if (ddi_prop_op(DDI_DEV_T_ANY, tgt_dip, PROP_LEN_AND_VAL_BUF,
1285                   DDI_PROP_DONTPASS | DDI_PROP_CANSLEEP, LIP_CNT_PROP,
1286                   (caddr_t)&lip_cnt, &t_len) != DDI_SUCCESS) {
1287         return (DDI_FAILURE);
1288     }
1289     /* and our LUN property */
1290     t_len = sizeof (lun);
1291     if (ddi_prop_op(DDI_DEV_T_ANY, tgt_dip, PROP_LEN_AND_VAL_BUF,
1292                   DDI_PROP_DONTPASS | DDI_PROP_CANSLEEP, "lun",
1293                   (caddr_t)&lun, &t_len) != DDI_SUCCESS) {
1294         return (DDI_FAILURE);
1295     }

1297     /* find the target structure for this instance */
1298     mutex_enter(&sf->sf_mutex);
1299     if ((target = sf_lookup_target(sf, wwn, lun)) == NULL) {
1300         mutex_exit(&sf->sf_mutex);
1301         return (DDI_FAILURE);
1302     }

1304     mutex_enter(&target->sft_mutex);
1305     if ((sf->sf_lip_cnt == lip_cnt) && !(target->sft_state
1306         & SF_TARGET_INIT_DONE)) {
1307         /*
1308          * set links between HBA transport and target structures
1309          * and set done flag
1310          */
1311         hba_tran->tran_tgt_private = target;
1312         target->sft_tran = hba_tran;

```

```

1313         target->sft_state |= SF_TARGET_INIT_DONE;
1314     } else {
1315         /* already initialized ?? */
1316         mutex_exit(&target->sft_mutex);
1317         mutex_exit(&sf->sf_mutex);
1318         return (DDI_FAILURE);
1319     }
1320     mutex_exit(&target->sft_mutex);
1321     mutex_exit(&sf->sf_mutex);
1322
1323     return (DDI_SUCCESS);
1324 }
1325
1326
1327 /*
1328  * called by transport to free a target
1329  */
1330 /* ARGSUSED */
1331 static void
1332 sf_scsi_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
1333                 scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
1334 {
1335     struct sf_target *target = hba_tran->tran_tgt_private;
1336
1337     if (target != NULL) {
1338         mutex_enter(&target->sft_mutex);
1339         target->sft_tran = NULL;
1340         target->sft_state &= ~SF_TARGET_INIT_DONE;
1341         mutex_exit(&target->sft_mutex);
1342     }
1343 }
1344
1345
1346 /*
1347  * allocator for non-std size cdb/pkt_private/status -- return TRUE iff
1348  * success, else return FALSE
1349  */
1350 /*ARGSUSED*/
1351 static int
1352 sf_pkt_alloc_extern(struct sf *sf, struct sf_pkt *cmd,
1353                    int tgtlen, int statuslen, int kf)
1354 {
1355     caddr_t scbp, tgt;
1356     int failure = FALSE;
1357     struct scsi_pkt *pkt = CMD2PKT(cmd);
1358
1359
1360     tgt = scbp = NULL;
1361
1362     if (tgtlen > PKT_PRIV_LEN) {
1363         if ((tgt = kmem_zalloc(tgtlen, kf)) == NULL) {
1364             failure = TRUE;
1365         } else {
1366             cmd->cmd_flags |= CFLAG_PRIVEXTERN;
1367             pkt->pkt_private = tgt;
1368         }
1369     }
1370     if (statuslen > EXTCMD_STATUS_SIZE) {
1371         if ((scbp = kmem_zalloc((size_t)statuslen, kf)) == NULL) {
1372             failure = TRUE;
1373         } else {
1374             cmd->cmd_flags |= CFLAG_SCBEXTERN;
1375             pkt->pkt_scbp = (opaque_t)scbp;
1376         }
1377     }
1378     if (failure) {

```

```

1379         sf_pkt_destroy_extern(sf, cmd);
1380     }
1381     return (failure);
1382 }
1383
1384
1385 /*
1386  * deallocator for non-std size cdb/pkt_private/status
1387  */
1388 static void
1389 sf_pkt_destroy_extern(struct sf *sf, struct sf_pkt *cmd)
1390 {
1391     struct scsi_pkt *pkt = CMD2PKT(cmd);
1392
1393     if (cmd->cmd_flags & CFLAG_FREE) {
1394         cmn_err(CE_PANIC,
1395              "sf_scsi_impl_pktfree: freeing free packet");
1396         /* NOTREACHED */
1397         /* NOTREACHED */
1398     }
1399     if (cmd->cmd_flags & CFLAG_SCBEXTERN) {
1400         kmem_free((caddr_t)pkt->pkt_scbp,
1401                  (size_t)cmd->cmd_scbp);
1402     }
1403     if (cmd->cmd_flags & CFLAG_PRIVEXTERN) {
1404         kmem_free((caddr_t)pkt->pkt_private,
1405                  (size_t)cmd->cmd_privlen);
1406     }
1407
1408     cmd->cmd_flags = CFLAG_FREE;
1409     kmem_cache_free(sf->sf_pkt_cache, (void *)cmd);
1410 }
1411
1412
1413 /*
1414  * create or initialize a SCSI packet -- called internally and
1415  * by the transport
1416  */
1417 static struct scsi_pkt *
1418 sf_scsi_init_pkt(struct scsi_address *ap, struct scsi_pkt *pkt,
1419                 struct buf *bp, int cmdlen, int statuslen, int tgtlen,
1420                 int flags, int (*callback)(), caddr_t arg)
1421 {
1422     int kf;
1423     int failure = FALSE;
1424     struct sf_pkt *cmd;
1425     struct sf *sf = ADDR2SF(ap);
1426     struct sf_target *target = ADDR2TARGET(ap);
1427     struct sf_pkt *new_cmd = NULL;
1428     struct fcnl_packet *fpkt;
1429     fc_frame_header_t *hp;
1430     struct fcp_cmd *fcmd;
1431
1432
1433     /*
1434      * If we've already allocated a pkt once,
1435      * this request is for dma allocation only.
1436      */
1437     if (pkt == NULL) {
1438
1439         /*
1440          * First step of sf_scsi_init_pkt: pkt allocation
1441          */
1442         if (cmdlen > FCP_CDB_SIZE) {
1443             return (NULL);
1444         }

```

```

1446         kf = (callback == SLEEP_FUNC)? KM_SLEEP: KM_NOSLEEP;
1448         if ((cmd = kmem_cache_alloc(sf->sf_pkt_cache, kf)) != NULL) {
1449             /*
1450              * Selective zeroing of the pkt.
1451              */
1453             cmd->cmd_flags = 0;
1454             cmd->cmd_forw = 0;
1455             cmd->cmd_back = 0;
1456             cmd->cmd_next = 0;
1457             cmd->cmd_pkt = (struct scsi_pkt *)((char *)cmd +
1458             sizeof (struct sf_pkt) + sizeof (struct
1459             fcal_packet));
1460             cmd->cmd_fp_pkt = (struct fcal_packet *)((char *)cmd +
1461             sizeof (struct sf_pkt));
1462             cmd->cmd_fp_pkt->fcal_pkt_private = (opaque_t)cmd;
1463             cmd->cmd_state = SF_STATE_IDLE;
1464             cmd->cmd_pkt->pkt_ha_private = (opaque_t)cmd;
1465             cmd->cmd_pkt->pkt_scbp = (opaque_t)cmd->cmd_scsi_scb;
1466             cmd->cmd_pkt->pkt_comp = NULL;
1467             cmd->cmd_pkt->pkt_flags = 0;
1468             cmd->cmd_pkt->pkt_time = 0;
1469             cmd->cmd_pkt->pkt_resid = 0;
1470             cmd->cmd_pkt->pkt_reason = 0;
1471             cmd->cmd_cdblen = (uchar_t)cmdlen;
1472             cmd->cmd_scblen = statuslen;
1473             cmd->cmd_privlen = tgtlen;
1474             cmd->cmd_pkt->pkt_address = *ap;
1476             /* zero pkt_private */
1477             (int *) (cmd->cmd_pkt->pkt_private =
1478             cmd->cmd_pkt_private);
1479             bzero((caddr_t)cmd->cmd_pkt->pkt_private,
1480             PKT_PRIV_LEN);
1481         } else {
1482             failure = TRUE;
1483         }
1485         if (failure ||
1486             (tgtlen > PKT_PRIV_LEN) ||
1487             (statuslen > EXTCMDS_STATUS_SIZE)) {
1488             if (!failure) {
1489                 /* need to allocate more space */
1490                 failure = sf_pkt_alloc_extern(sf, cmd,
1491                 tgtlen, statuslen, kf);
1492             }
1493             if (failure) {
1494                 return (NULL);
1495             }
1496         }
1498         fpkt = cmd->cmd_fp_pkt;
1499         if (cmd->cmd_block == NULL) {
1501             /* allocate cmd/response pool buffers */
1502             if (sf_cr_alloc(sf, cmd, callback) == DDI_FAILURE) {
1503                 sf_pkt_destroy_extern(sf, cmd);
1504                 return (NULL);
1505             }
1507             /* fill in the FC-AL packet */
1508             fpkt->fcal_pkt_cookie = sf->sf_socp;
1509             fpkt->fcal_pkt_comp = sf_cmd_callback;
1510             fpkt->fcal_pkt_flags = 0;

```

```

1511         fpkt->fcal_magic = FCALP_MAGIC;
1512         fpkt->fcal_social_request.sr_soc_hdr.sh_flags =
1513         (ushort_t)(SOC_FC_HEADER |
1514         sf->sf_sochandle->fcal_portno);
1515         fpkt->fcal_social_request.sr_soc_hdr.sh_class = 3;
1516         fpkt->fcal_social_request.sr_cqhdr.cq_hdr_count = 1;
1517         fpkt->fcal_social_request.sr_cqhdr.cq_hdr_flags = 0;
1518         fpkt->fcal_social_request.sr_cqhdr.cq_hdr_seqno = 0;
1519         fpkt->fcal_social_request.sr_dataseg[0].fc_base =
1520         (uint32_t)cmd->cmd_dmac;
1521         fpkt->fcal_social_request.sr_dataseg[0].fc_count =
1522         sizeof (struct fcp_cmd);
1523         fpkt->fcal_social_request.sr_dataseg[1].fc_base =
1524         (uint32_t)cmd->cmd_rsp_dmac;
1525         fpkt->fcal_social_request.sr_dataseg[1].fc_count =
1526         FCP_MAX_RSP_IU_SIZE;
1528         /* Fill in the Fabric Channel Header */
1529         hp = &fpkt->fcal_social_request.sr_fc_frame_hdr;
1530         hp->r_ctl = R_CTL_COMMAND;
1531         hp->type = TYPE_SCSI_FCP;
1532         hp->f_ctl = F_CTL_SEQ_INITIATIVE | F_CTL_FIRST_SEQ;
1533         hp->reserved1 = 0;
1534         hp->seq_id = 0;
1535         hp->df_ctl = 0;
1536         hp->seq_cnt = 0;
1537         hp->ox_id = 0xffff;
1538         hp->rx_id = 0xffff;
1539         hp->ro = 0;
1541         /* Establish the LUN */
1542         bcopy((caddr_t)&target->sft_lun.b,
1543         (caddr_t)&cmd->cmd_block->fcp_ent_addr,
1544         FCP_LUN_SIZE);
1545         *((int32_t *)&cmd->cmd_block->fcp_cntl) = 0;
1546     }
1547     cmd->cmd_pkt->pkt_cdbp = cmd->cmd_block->fcp_cdb;
1549     mutex_enter(&target->sft_pkt_mutex);
1551     target->sft_pkt_tail->cmd_forw = cmd;
1552     cmd->cmd_back = target->sft_pkt_tail;
1553     cmd->cmd_forw = (struct sf_pkt *)&target->sft_pkt_head;
1554     target->sft_pkt_tail = cmd;
1556     mutex_exit(&target->sft_pkt_mutex);
1557     new_cmd = cmd; /* for later cleanup if needed */
1558 } else {
1559     /* pkt already exists -- just a request for DMA allocation */
1560     cmd = PKT2CMD(pkt);
1561     fpkt = cmd->cmd_fp_pkt;
1562 }
1564 /* zero cdb (bzero is too slow) */
1565 bzero((caddr_t)cmd->cmd_pkt->pkt_cdbp, cmdlen);
1567 /*
1568  * Second step of sf_scsi_init_pkt: dma allocation
1569  * Set up dma info
1570  */
1571 if ((bp != NULL) && (bp->b_bcount != 0)) {
1572     int cmd_flags, dma_flags;
1573     int rval = 0;
1574     uint_t dmacookie_count;
1576     /* there is a buffer and some data to transfer */

```

```

1578     /* set up command and DMA flags */
1579     cmd_flags = cmd->cmd_flags;
1580     if (bp->b_flags & B_READ) {
1581         /* a read */
1582         cmd_flags &= ~CFLAG_DMASEND;
1583         dma_flags = DDI_DMA_READ;
1584     } else {
1585         /* a write */
1586         cmd_flags |= CFLAG_DMASEND;
1587         dma_flags = DDI_DMA_WRITE;
1588     }
1589     if (flags & PKT_CONSISTENT) {
1590         cmd_flags |= CFLAG_CMDIOPB;
1591         dma_flags |= DDI_DMA_CONSISTENT;
1592     }
1593
1594     /* ensure we have a DMA handle */
1595     if (cmd->cmd_dmahandle == NULL) {
1596         rval = ddi_dma_alloc_handle(sf->sf_dip,
1597             sf->sf_sochandle->fcal_dmaattr, callback, arg,
1598             &cmd->cmd_dmahandle);
1599     }
1600
1601     if (rval == 0) {
1602         /* bind our DMA handle to our buffer */
1603         rval = ddi_dma_buf_bind_handle(cmd->cmd_dmahandle, bp,
1604             dma_flags, callback, arg, &cmd->cmd_dmacookie,
1605             &dmacookie_count);
1606     }
1607
1608     if (rval != 0) {
1609         /* DMA failure */
1610         SF_DEBUG(2, (sf, CE_CONT, "ddi_dma_buf.. failed\n"));
1611         switch (rval) {
1612             case DDI_DMA_NORESOURCES:
1613                 bioerror(bp, 0);
1614                 break;
1615             case DDI_DMA_BADATTR:
1616             case DDI_DMA_NOMAPPING:
1617                 bioerror(bp, EFAULT);
1618                 break;
1619             case DDI_DMA_TOOBIG:
1620             default:
1621                 bioerror(bp, EINVAL);
1622                 break;
1623         }
1624         /* clear valid flag */
1625         cmd->cmd_flags = cmd_flags & ~CFLAG_DMAVALID;
1626         if (new_cmd != NULL) {
1627             /* destroy packet if we just created it */
1628             sf_scsi_destroy_pkt(ap, new_cmd->cmd_pkt);
1629         }
1630         return (NULL);
1631     }
1632
1633     ASSERT(dmacookie_count == 1);
1634     /* set up amt to transfer and set valid flag */
1635     cmd->cmd_dmacount = bp->b_bcount;
1636     cmd->cmd_flags = cmd_flags | CFLAG_DMAVALID;
1637
1638     ASSERT(cmd->cmd_dmahandle != NULL);
1639 }
1640
1641 /* set up FC-AL packet */
1642 fcmd = cmd->cmd_block;

```

```

1644     if (cmd->cmd_flags & CFLAG_DMAVALID) {
1645         if (cmd->cmd_flags & CFLAG_DMASEND) {
1646             /* DMA write */
1647             fcmd->fcp_cntl.cntl_read_data = 0;
1648             fcmd->fcp_cntl.cntl_write_data = 1;
1649             fpkt->fcal_social_request.sr_cqhdr.cq_hdr_type =
1650                 CQ_TYPE_IO_WRITE;
1651         } else {
1652             /* DMA read */
1653             fcmd->fcp_cntl.cntl_read_data = 1;
1654             fcmd->fcp_cntl.cntl_write_data = 0;
1655             fpkt->fcal_social_request.sr_cqhdr.cq_hdr_type =
1656                 CQ_TYPE_IO_READ;
1657         }
1658         fpkt->fcal_social_request.sr_dataseg[2].fc_base =
1659             (uint32_t)cmd->cmd_dmacookie.dmac_address;
1660         fpkt->fcal_social_request.sr_dataseg[2].fc_count =
1661             cmd->cmd_dmacookie.dmac_size;
1662         fpkt->fcal_social_request.sr_soc_hdr.sh_seg_cnt = 3;
1663         fpkt->fcal_social_request.sr_soc_hdr.sh_byte_cnt =
1664             cmd->cmd_dmacookie.dmac_size;
1665         fcmd->fcp_data_len = cmd->cmd_dmacookie.dmac_size;
1666     } else {
1667         /* not a read or write */
1668         fcmd->fcp_cntl.cntl_read_data = 0;
1669         fcmd->fcp_cntl.cntl_write_data = 0;
1670         fpkt->fcal_social_request.sr_cqhdr.cq_hdr_type = CQ_TYPE_SIMPLE;
1671         fpkt->fcal_social_request.sr_soc_hdr.sh_seg_cnt = 2;
1672         fpkt->fcal_social_request.sr_soc_hdr.sh_byte_cnt =
1673             sizeof (struct fcp_cmd);
1674         fcmd->fcp_data_len = 0;
1675     }
1676     fcmd->fcp_cntl.cntl_qtype = FCP_QTYPE_SIMPLE;
1677
1678     return (cmd->cmd_pkt);
1679 }
1680
1681 /*
1682  * destroy a SCSI packet -- called internally and by the transport
1683  */
1684 static void
1685 sf_scsi_destroy_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
1686 {
1687     struct sf_pkt *cmd = PKT2CMD(pkt);
1688     struct sf *sf = ADDR2SF(ap);
1689     struct sf_target *target = ADDR2TARGET(ap);
1690     struct fcal_packet *fpkt = cmd->cmd_fp_pkt;
1691
1692     if (cmd->cmd_flags & CFLAG_DMAVALID) {
1693         /* DMA was set up -- clean up */
1694         (void) ddi_dma_unbind_handle(cmd->cmd_dmahandle);
1695         cmd->cmd_flags ^= CFLAG_DMAVALID;
1696     }
1697
1698     /* take this packet off the doubly-linked list */
1699     mutex_enter(&target->sft_pkt_mutex);
1700     cmd->cmd_back->cmd_forw = cmd->cmd_forw;
1701     cmd->cmd_forw->cmd_back = cmd->cmd_back;
1702     mutex_exit(&target->sft_pkt_mutex);
1703
1704     fpkt->fcal_pkt_flags = 0;
1705     /* free the packet */
1706     if ((cmd->cmd_flags &

```



```

1709         (CFLAG_FREE | CFLAG_PRIVEXTERN | CFLAG_SCBEXTERN)) == 0) {
1710             /* just a regular packet */
1711             ASSERT(cmd->cmd_state != SF_STATE_ISSUED);
1712             cmd->cmd_flags = CFLAG_FREE;
1713             kmem_cache_free(sf->sf_pkt_cache, (void *)cmd);
1714         } else {
1715             /* a packet with extra memory */
1716             sf_pkt_destroy_extern(sf, cmd);
1717         }
1718     }

1721 /*
1722  * called by transport to unbind DMA handle
1723  */
1724 /* ARGSUSED */
1725 static void
1726 sf_scsi_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt)
1727 {
1728     struct sf_pkt *cmd = PKT2CMD(pkt);

1731     if (cmd->cmd_flags & CFLAG_DMAVALID) {
1732         (void) ddi_dma_unbind_handle(cmd->cmd_dmahandle);
1733         cmd->cmd_flags ^= CFLAG_DMAVALID;
1734     }

1736 }

1739 /*
1740  * called by transport to synchronize CPU and I/O views of memory
1741  */
1742 /* ARGSUSED */
1743 static void
1744 sf_scsi_sync_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
1745 {
1746     struct sf_pkt *cmd = PKT2CMD(pkt);

1749     if (cmd->cmd_flags & CFLAG_DMAVALID) {
1750         if (ddi_dma_sync(cmd->cmd_dmahandle, (off_t)0, (size_t)0,
1751             (cmd->cmd_flags & CFLAG_DMASEND) ?
1752             DDI_DMA_SYNC_FORDEV : DDI_DMA_SYNC_FORCPU) !=
1753             DDI_SUCCESS) {
1754             cmn_err(CE_WARN, "sf: sync pkt failed");
1755         }
1756     }
1757 }

1760 /*
1761  * routine for reset notification setup, to register or cancel. -- called
1762  * by transport
1763  */
1764 static int
1765 sf_scsi_reset_notify(struct scsi_address *ap, int flag,
1766     void (*callback)(caddr_t), caddr_t arg)
1767 {
1768     struct sf         *sf = ADDR2SF(ap);

1770     return (scsi_hba_reset_notify_setup(ap, flag, callback, arg,
1771         &sf->sf_mutex, &sf->sf_reset_notify_listf));
1772 }

```

```

1775 /*
1776  * called by transport to get port WWN property (except sun4u)
1777  */
1778 /* ARGSUSED */
1779 static int
1780 sf_scsi_get_name(struct scsi_device *sd, char *name, int len)
1781 {
1782     char tbuf[(FC_WWN_SIZE*2)+1];
1783     unsigned char wwn[FC_WWN_SIZE];
1784     int i, lun;
1785     dev_info_t *tgt_dip;

1787     tgt_dip = sd->sd_dev;
1788     i = sizeof (wwn);
1789     if (ddi_prop_op(DDI_DEV_T_ANY, tgt_dip, PROP_LEN_AND_VAL_BUF,
1790         DDI_PROP_DONTPASS | DDI_PROP_CANSLEEP, PORT_WWN_PROP,
1791         (caddr_t)&wwn, &i) != DDI_SUCCESS) {
1792         name[0] = '\0';
1793         return (0);
1794     }
1795     i = sizeof (lun);
1796     if (ddi_prop_op(DDI_DEV_T_ANY, tgt_dip, PROP_LEN_AND_VAL_BUF,
1797         DDI_PROP_DONTPASS | DDI_PROP_CANSLEEP, "lun",
1798         (caddr_t)&lun, &i) != DDI_SUCCESS) {
1799         name[0] = '\0';
1800         return (0);
1801     }
1802     for (i = 0; i < FC_WWN_SIZE; i++)
1803         (void) sprintf(&tbuf[i << 1], "%02x", wwn[i]);
1804     (void) sprintf(name, "w%s,%x", tbuf, lun);
1805     return (1);
1806 }

1809 /*
1810  * called by transport to get target soft AL-PA (except sun4u)
1811  */
1812 /* ARGSUSED */
1813 static int
1814 sf_scsi_get_bus_addr(struct scsi_device *sd, char *name, int len)
1815 {
1816     struct sf_target *target = ADDR2TARGET(&sd->sd_address);

1818     if (target == NULL)
1819         return (0);

1821     (void) sprintf(name, "%x", target->sft_al_pa);
1822     return (1);
1823 }

1826 /*
1827  * add to the command/response buffer pool for this sf instance
1828  */
1829 static int
1830 sf_add_cr_pool(struct sf *sf)
1831 {
1832     int             cmd_buf_size;
1833     size_t         real_cmd_buf_size;
1834     int             rsp_buf_size;
1835     size_t         real_rsp_buf_size;
1836     uint_t         i, ccount;
1837     struct sf_cr_pool *ptr;
1838     struct sf_cr_free_elem *cptr;
1839     caddr_t        dptr, eptr;
1840     ddi_dma_cookie_t cmd_cookie;

```

```

1841     ddi_dma_cookie_t     rsp_cookie;
1842     int                   cmd_bound = FALSE, rsp_bound = FALSE;

1845     /* allocate room for the pool */
1846     if ((ptr = kmem_zalloc(sizeof (struct sf_cr_pool), KM_NOSLEEP)) ==
1847         NULL) {
1848         return (DDI_FAILURE);
1849     }

1851     /* allocate a DMA handle for the command pool */
1852     if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
1853         DDI_DMA_DONTWAIT, NULL, &ptr->cmd_dma_handle) != DDI_SUCCESS) {
1854         goto fail;
1855     }

1857     /*
1858     * Get a piece of memory in which to put commands
1859     */
1860     cmd_buf_size = (sizeof (struct fcp_cmd) * SF_ELEMS_IN_POOL + 7) & ~7;
1861     if (ddi_dma_mem_alloc(ptr->cmd_dma_handle, cmd_buf_size,
1862         sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
1863         DDI_DMA_DONTWAIT, NULL, (caddr_t *)&ptr->cmd_base,
1864         &real_cmd_buf_size, &ptr->cmd_acc_handle) != DDI_SUCCESS) {
1865         goto fail;
1866     }

1868     /* bind the DMA handle to an address */
1869     if (ddi_dma_addr_bind_handle(ptr->cmd_dma_handle, NULL,
1870         ptr->cmd_base, real_cmd_buf_size,
1871         DDI_DMA_WRITE | DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT,
1872         NULL, &cmd_cookie, &ccount) != DDI_DMA_MAPPED) {
1873         goto fail;
1874     }
1875     cmd_bound = TRUE;
1876     /* ensure only one cookie was allocated */
1877     if (ccount != 1) {
1878         goto fail;
1879     }

1881     /* allocate a DMA handle for the response pool */
1882     if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
1883         DDI_DMA_DONTWAIT, NULL, &ptr->rsp_dma_handle) != DDI_SUCCESS) {
1884         goto fail;
1885     }

1887     /*
1888     * Get a piece of memory in which to put responses
1889     */
1890     rsp_buf_size = FCP_MAX_RSP_IU_SIZE * SF_ELEMS_IN_POOL;
1891     if (ddi_dma_mem_alloc(ptr->rsp_dma_handle, rsp_buf_size,
1892         sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
1893         DDI_DMA_DONTWAIT, NULL, (caddr_t *)&ptr->rsp_base,
1894         &real_rsp_buf_size, &ptr->rsp_acc_handle) != DDI_SUCCESS) {
1895         goto fail;
1896     }

1898     /* bind the DMA handle to an address */
1899     if (ddi_dma_addr_bind_handle(ptr->rsp_dma_handle, NULL,
1900         ptr->rsp_base, real_rsp_buf_size,
1901         DDI_DMA_READ | DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT,
1902         NULL, &rsp_cookie, &ccount) != DDI_DMA_MAPPED) {
1903         goto fail;
1904     }
1905     rsp_bound = TRUE;
1906     /* ensure only one cookie was allocated */

```

```

1907     if (ccount != 1) {
1908         goto fail;
1909     }

1911     /*
1912     * Generate a (cmd/rsp structure) free list
1913     */
1914     /* ensure ptr points to start of long word (8-byte block) */
1915     dptr = (caddr_t)((uintptr_t)(ptr->cmd_base) + 7 & ~7);
1916     /* keep track of actual size after moving pointer */
1917     real_cmd_buf_size -= (dptr - ptr->cmd_base);
1918     eptr = ptr->rsp_base;

1920     /* set actual total number of entries */
1921     ptr->ntot = min((real_cmd_buf_size / sizeof (struct fcp_cmd)),
1922         (real_rsp_buf_size / FCP_MAX_RSP_IU_SIZE));
1923     ptr->nfree = ptr->ntot;
1924     ptr->free = (struct sf_cr_free_elem *)ptr->cmd_base;
1925     ptr->sf = sf;

1927     /* set up DMA for each pair of entries */
1928     i = 0;
1929     while (i < ptr->ntot) {
1930         cptr = (struct sf_cr_free_elem *)dptr;
1931         dptr += sizeof (struct fcp_cmd);

1933         cptr->next = (struct sf_cr_free_elem *)dptr;
1934         cptr->rsp = eptr;

1936         cptr->cmd_dmac = cmd_cookie.dmac_address +
1937             (uint32_t)((caddr_t)cptr - ptr->cmd_base);

1939         cptr->rsp_dmac = rsp_cookie.dmac_address +
1940             (uint32_t)((caddr_t)eptr - ptr->rsp_base);

1942         eptr += FCP_MAX_RSP_IU_SIZE;
1943         i++;
1944     }

1946     /* terminate the list */
1947     cptr->next = NULL;

1949     /* add this list at front of current one */
1950     mutex_enter(&sf->sf_cr_mutex);
1951     ptr->next = sf->sf_cr_pool;
1952     sf->sf_cr_pool = ptr;
1953     sf->sf_cr_pool_cnt++;
1954     mutex_exit(&sf->sf_cr_mutex);

1956     return (DDI_SUCCESS);

1958 fail:
1959     /* we failed so clean up */
1960     if (ptr->cmd_dma_handle != NULL) {
1961         if (cmd_bound) {
1962             (void) ddi_dma_unbind_handle(ptr->cmd_dma_handle);
1963         }
1964         ddi_dma_free_handle(&ptr->cmd_dma_handle);
1965     }

1967     if (ptr->rsp_dma_handle != NULL) {
1968         if (rsp_bound) {
1969             (void) ddi_dma_unbind_handle(ptr->rsp_dma_handle);
1970         }
1971         ddi_dma_free_handle(&ptr->rsp_dma_handle);
1972     }

```

```

1974     if (ptr->cmd_base != NULL) {
1975         ddi_dma_mem_free(&ptr->cmd_acc_handle);
1976     }

1978     if (ptr->rsp_base != NULL) {
1979         ddi_dma_mem_free(&ptr->rsp_acc_handle);
1980     }

1982     kmem_free((caddr_t)ptr, sizeof (struct sf_cr_pool));
1983     return (DDI_FAILURE);
1984 }

1987 /*
1988  * allocate a command/response buffer from the pool, allocating more
1989  * in the pool as needed
1990  */
1991 static int
1992 sf_cr_alloc(struct sf *sf, struct sf_pkt *cmd, int (*func)())
1993 {
1994     struct sf_cr_pool *ptr;
1995     struct sf_cr_free_elem *cptr;

1998     mutex_enter(&sf->sf_cr_mutex);

2000 try_again:

2002     /* find a free buffer in the existing pool */
2003     ptr = sf->sf_cr_pool;
2004     while (ptr != NULL) {
2005         if (ptr->nfree != 0) {
2006             ptr->nfree--;
2007             break;
2008         } else {
2009             ptr = ptr->next;
2010         }
2011     }

2013     /* did we find a free buffer ? */
2014     if (ptr != NULL) {
2015         /* we found a free buffer -- take it off the free list */
2016         cptr = ptr->free;
2017         ptr->free = cptr->next;
2018         mutex_exit(&sf->sf_cr_mutex);
2019         /* set up the command to use the buffer pair */
2020         cmd->cmd_block = (struct fcp_cmd *)cptr;
2021         cmd->cmd_dmac = cptr->cmd_dmac;
2022         cmd->cmd_rsp_dmac = cptr->rsp_dmac;
2023         cmd->cmd_rsp_block = (struct fcp_rsp *)cptr->rsp;
2024         cmd->cmd_cr_pool = ptr;
2025         return (DDI_SUCCESS);          /* success */
2026     }

2028     /* no free buffer available -- can we allocate more ? */
2029     if (sf->sf_cr_pool_cnt < SF_CR_POOL_MAX) {
2030         /* we need to allocate more buffer pairs */
2031         if (sf->sf_cr_flag) {
2032             /* somebody already allocating for this instance */
2033             if (func == SLEEP_FUNC) {
2034                 /* user wants to wait */
2035                 cv_wait(&sf->sf_cr_cv, &sf->sf_cr_mutex);
2036                 /* we've been woken so go try again */
2037                 goto try_again;
2038             }

```

```

2039         /* user does not want to wait */
2040         mutex_exit(&sf->sf_cr_mutex);
2041         sf->sf_stats.cralloc_failures++;
2042         return (DDI_FAILURE);        /* give up */
2043     }
2044     /* set flag saying we're allocating */
2045     sf->sf_cr_flag = 1;
2046     mutex_exit(&sf->sf_cr_mutex);
2047     /* add to our pool */
2048     if (sf_add_cr_pool(sf) != DDI_SUCCESS) {
2049         /* couldn't add to our pool for some reason */
2050         mutex_enter(&sf->sf_cr_mutex);
2051         sf->sf_cr_flag = 0;
2052         cv_broadcast(&sf->sf_cr_cv);
2053         mutex_exit(&sf->sf_cr_mutex);
2054         sf->sf_stats.cralloc_failures++;
2055         return (DDI_FAILURE);        /* give up */
2056     }
2057     /*
2058      * clear flag saying we're allocating and tell all other
2059      * that care
2060      */
2061     mutex_enter(&sf->sf_cr_mutex);
2062     sf->sf_cr_flag = 0;
2063     cv_broadcast(&sf->sf_cr_cv);
2064     /* now that we have more buffers try again */
2065     goto try_again;
2066 }

2068     /* we don't have room to allocate any more buffers */
2069     mutex_exit(&sf->sf_cr_mutex);
2070     sf->sf_stats.cralloc_failures++;
2071     return (DDI_FAILURE);          /* give up */
2072 }

2075 /*
2076  * free a cmd/response buffer pair in our pool
2077  */
2078 static void
2079 sf_cr_free(struct sf_cr_pool *cp, struct sf_pkt *cmd)
2080 {
2081     struct sf *sf = cp->sf;
2082     struct sf_cr_free_elem *elem;

2084     elem = (struct sf_cr_free_elem *)cmd->cmd_block;
2085     elem->rsp = (caddr_t)cmd->cmd_rsp_block;
2086     elem->cmd_dmac = cmd->cmd_dmac;
2087     elem->rsp_dmac = cmd->cmd_rsp_dmac;

2089     mutex_enter(&sf->sf_cr_mutex);
2090     cp->nfree++;
2091     ASSERT(cp->nfree <= cp->ntot);

2093     elem->next = cp->free;
2094     cp->free = elem;
2095     mutex_exit(&sf->sf_cr_mutex);
2096 }

2099 /*
2100  * free our pool of cmd/response buffers
2101  */
2102 static void
2103 sf_crpool_free(struct sf *sf)
2104 {

```

```

2105     struct sf_cr_pool *cp, *prev;

2107     prev = NULL;
2108     mutex_enter(&sf->sf_cr_mutex);
2109     cp = sf->sf_cr_pool;
2110     while (cp != NULL) {
2111         if (cp->nfree == cp->ntot) {
2112             if (prev != NULL) {
2113                 prev->next = cp->next;
2114             } else {
2115                 sf->sf_cr_pool = cp->next;
2116             }
2117             sf->sf_cr_pool_cnt--;
2118             mutex_exit(&sf->sf_cr_mutex);

2120             (void) ddi_dma_unbind_handle(cp->cmd_dma_handle);
2121             ddi_dma_free_handle(&cp->cmd_dma_handle);
2122             (void) ddi_dma_unbind_handle(cp->rsp_dma_handle);
2123             ddi_dma_free_handle(&cp->rsp_dma_handle);
2124             ddi_dma_mem_free(&cp->cmd_acc_handle);
2125             ddi_dma_mem_free(&cp->rsp_acc_handle);
2126             kmem_free((caddr_t)cp, sizeof (struct sf_cr_pool));
2127             return;
2128         }
2129         prev = cp;
2130         cp = cp->next;
2131     }
2132     mutex_exit(&sf->sf_cr_mutex);
2133 }

2136 /* ARGSUSED */
2137 static int
2138 sf_kmem_cache_constructor(void *buf, void *arg, int size)
2139 {
2140     struct sf_pkt *cmd = buf;

2142     mutex_init(&cmd->cmd_abort_mutex, NULL, MUTEX_DRIVER, NULL);
2143     cmd->cmd_block = NULL;
2144     cmd->cmd_dmahandle = NULL;
2145     return (0);
2146 }

2149 /* ARGSUSED */
2150 static void
2151 sf_kmem_cache_destructor(void *buf, void *size)
2152 {
2153     struct sf_pkt *cmd = buf;

2155     if (cmd->cmd_dmahandle != NULL) {
2156         ddi_dma_free_handle(&cmd->cmd_dmahandle);
2157     }

2159     if (cmd->cmd_block != NULL) {
2160         sf_cr_free(cmd->cmd_cr_pool, cmd);
2161     }
2162     mutex_destroy(&cmd->cmd_abort_mutex);
2163 }

2166 /*
2167  * called by transport when a state change occurs
2168  */
2169 static void
2170 sf_statec_callback(void *arg, int msg)

```

```

2171 {
2172     struct sf *sf = (struct sf *)arg;
2173     struct sf_target *target;
2174     int i;
2175     struct sf_pkt *cmd;
2176     struct scsi_pkt *pkt;

2180     switch (msg) {

2182     case FCAL_STATUS_LOOP_ONLINE: {
2183         uchar_t al_pa; /* to save AL-PA */
2184         int ret; /* ret value from getmap */
2185         int lip_cnt; /* to save current count */
2186         int cnt; /* map length */

2188         /*
2189          * the loop has gone online
2190          */
2191         SF_DEBUG(1, (sf, CE_CONT, "sf%d: loop online\n",
2192             ddi_get_instance(sf->sf_dip)));
2193         mutex_enter(&sf->sf_mutex);
2194         sf->sf_lip_cnt++;
2195         sf->sf_state = SF_STATE_ONLINING;
2196         mutex_exit(&sf->sf_mutex);

2198         /* scan each target hash queue */
2199         for (i = 0; i < SF_NUM_HASH_QUEUES; i++) {
2200             target = sf->sf_wm_lists[i];
2201             while (target != NULL) {
2202                 /*
2203                  * foreach target, if it's not offline then
2204                  * mark it as busy
2205                  */
2206                 mutex_enter(&target->sft_mutex);
2207                 if (!(target->sft_state & SF_TARGET_OFFLINE))
2208                     target->sft_state |= (SF_TARGET_BUSY
2209                         | SF_TARGET_MARK);
2210             #ifdef DEBUG
2211                 /*
2212                  * for debugging, print out info on any
2213                  * pending commands (left hanging)
2214                  */
2215                 cmd = target->sft_pkt_head;
2216                 while (cmd != (struct sf_pkt *)&target->
2217                     sft_pkt_head) {
2218                     if (cmd->cmd_state ==
2219                         SF_STATE_ISSUED) {
2220                         SF_DEBUG(1, (sf, CE_CONT,
2221                             "cmd 0x%p pending "
2222                             "after lip\n",
2223                             (void *)cmd->cmd_fp_pkt));
2224                     }
2225                     cmd = cmd->cmd_forw;
2226                 }
2227             #endif
2228                 mutex_exit(&target->sft_mutex);
2229                 target = target->sft_next;
2230             }
2231         }

2233         /*
2234          * since the loop has just gone online get a new map from
2235          * the transport
2236          */

```

```

2237     if ((ret = soc_get_lilp_map(sf->sf_sochandle, sf->sf_socp,
2238         sf->sf_sochandle->fcal_portno, (uint32_t)sf->
2239         sf_lilp_dmacookie.dmac_address, 1)) != FCAL_SUCCESS) {
2240         if (sf_core && (sf_core & SF_CORE_LILP_FAILED)) {
2241             (void) soc_take_core(sf->sf_sochandle,
2242                 sf->sf_socp);
2243             sf_core = 0;
2244         }
2245         sf_log(sf, CE_WARN,
2246             "!soc lilp map failed status=0x%x\n", ret);
2247         mutex_enter(&sf->sf_mutex);
2248         sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
2249         sf->sf_lip_cnt++;
2250         sf->sf_state = SF_STATE_OFFLINE;
2251         mutex_exit(&sf->sf_mutex);
2252         return;
2253     }
2254
2255     /* ensure consistent view of DMA memory */
2256     (void) ddi_dma_sync(sf->sf_lilp_dmahandle, (off_t)0, (size_t)0,
2257         DDI_DMA_SYNC_FORKERNEL);
2258
2259     /* how many entries in map ? */
2260     cnt = sf->sf_lilp_map->lilp_length;
2261     if (cnt >= SF_MAX_LILP_ENTRIES) {
2262         sf_log(sf, CE_WARN, "invalid lilp map\n");
2263         return;
2264     }
2265
2266     mutex_enter(&sf->sf_mutex);
2267     sf->sf_device_count = cnt - 1;
2268     sf->sf_al_pa = sf->sf_lilp_map->lilp_myalpa;
2269     lip_cnt = sf->sf_lip_cnt;
2270     al_pa = sf->sf_al_pa;
2271
2272     SF_DEBUG(1, (sf, CE_CONT,
2273         "!lilp map has %d entries, al_pa is %x\n", cnt, al_pa));
2274
2275     /*
2276     * since the last entry of the map may be mine (common) check
2277     * for that, and if it is we have one less entry to look at
2278     */
2279     if (sf->sf_lilp_map->lilp_alpalist[cnt-1] == al_pa) {
2280         cnt--;
2281     }
2282     /* If we didn't get a valid loop map enable all targets */
2283     if (sf->sf_lilp_map->lilp_magic == FCAL_BADLILP_MAGIC) {
2284         for (i = 0; i < sizeof (sf_switch_to_alpa); i++)
2285             sf->sf_lilp_map->lilp_alpalist[i] =
2286                 sf_switch_to_alpa[i];
2287         cnt = i;
2288         sf->sf_device_count = cnt - 1;
2289     }
2290     if (sf->sf_device_count == 0) {
2291         sf_finish_init(sf, lip_cnt);
2292         mutex_exit(&sf->sf_mutex);
2293         break;
2294     }
2295     mutex_exit(&sf->sf_mutex);
2296
2297     SF_DEBUG(2, (sf, CE_WARN,
2298         "!stateg callback: starting with %d targets\n",
2299         sf->sf_device_count));
2300
2301     /* scan loop map, logging into all ports (except mine) */
2302     for (i = 0; i < cnt; i++) {

```

```

2303         SF_DEBUG(1, (sf, CE_CONT,
2304             "!lilp map entry %d = %x,%x\n", i,
2305             sf->sf_lilp_map->lilp_alpalist[i],
2306             sf_alpa_to_switch[
2307                 sf->sf_lilp_map->lilp_alpalist[i]]);
2308         /* is this entry for somebody else ? */
2309         if (sf->sf_lilp_map->lilp_alpalist[i] != al_pa) {
2310             /* do a PLOGI to this port */
2311             if (!sf_login(sf, LA_ELS_PLOGI,
2312                 sf->sf_lilp_map->lilp_alpalist[i],
2313                 sf->sf_lilp_map->lilp_alpalist[cnt-1],
2314                 lip_cnt)) {
2315                 /* a problem logging in */
2316                 mutex_enter(&sf->sf_mutex);
2317                 if (lip_cnt == sf->sf_lip_cnt) {
2318                     /*
2319                      * problem not from a new LIP
2320                      */
2321                     sf->sf_device_count--;
2322                     ASSERT(sf->sf_device_count
2323                         >= 0);
2324                     if (sf->sf_device_count == 0) {
2325                         sf_finish_init(sf,
2326                             lip_cnt);
2327                     }
2328                 }
2329                 mutex_exit(&sf->sf_mutex);
2330             }
2331         }
2332     }
2333     break;
2334 }
2335
2336 case FCAL_STATUS_ERR_OFFLINE:
2337     /*
2338     * loop has gone offline due to an error
2339     */
2340     SF_DEBUG(1, (sf, CE_CONT, "sf%d: loop offline\n",
2341         ddi_get_instance(sf->sf_dip)));
2342     mutex_enter(&sf->sf_mutex);
2343     sf->sf_lip_cnt++;
2344     sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
2345     if (!sf->sf_online_timer) {
2346         sf->sf_online_timer = sf_watchdog_time +
2347             SF_ONLINE_TIMEOUT;
2348     }
2349     /*
2350     * if we are suspended, preserve the SF_STATE_SUSPENDED flag,
2351     * since throttling logic in sf_watch() depends on
2352     * preservation of this flag while device is suspended
2353     */
2354     if (sf->sf_state & SF_STATE_SUSPENDED) {
2355         sf->sf_state |= SF_STATE_OFFLINE;
2356         SF_DEBUG(1, (sf, CE_CONT,
2357             "sf_stateg_callback, sf%d: "
2358             "got FCAL_STATUS_OFFLINE during DDI_SUSPEND\n",
2359             ddi_get_instance(sf->sf_dip)));
2360     } else {
2361         sf->sf_state = SF_STATE_OFFLINE;
2362     }
2363
2364     /* scan each possible target on the loop */
2365     for (i = 0; i < sf_max_targets; i++) {
2366         target = sf->sf_targets[i];
2367         while (target != NULL) {
2368             mutex_enter(&target->sft_mutex);

```

```

2369         if (!(target->sft_state & SF_TARGET_OFFLINE))
2370             target->sft_state |= (SF_TARGET_BUSY
2371                 | SF_TARGET_MARK);
2372         mutex_exit(&target->sft_mutex);
2373         target = target->sft_next_lun;
2374     }
2375 }
2376 mutex_exit(&sf->sft_mutex);
2377 break;
2378
2379 case FCAL_STATE_RESET: {
2380     struct sf_els_hdr    *privp; /* ptr to private list */
2381     struct sf_els_hdr    *tmpp1; /* tmp prev hdr ptr */
2382     struct sf_els_hdr    *tmpp2; /* tmp next hdr ptr */
2383     struct sf_els_hdr    *head; /* to save our private list */
2384     struct fcml_packet    *fpkt; /* ptr to pkt in hdr */
2385
2386     /*
2387      * a transport reset
2388      */
2389     SF_DEBUG(1, (sf, CE_CONT, "!\sf%d: soc reset\n",
2390         ddi_get_instance(sf->sft_dip)));
2391     tmpp1 = head = NULL;
2392     mutex_enter(&sf->sft_mutex);
2393     sf->sft_lip_cnt++;
2394     sf->sft_timer = sf_watchdog_time + SF_RESET_TIMEOUT;
2395     /*
2396      * if we are suspended, preserve the SF_STATE_SUSPENDED flag,
2397      * since throttling logic in sf_watch() depends on
2398      * preservation of this flag while device is suspended
2399      */
2400     if (sf->sft_state & SF_STATE_SUSPENDED) {
2401         sf->sft_state |= SF_STATE_OFFLINE;
2402         SF_DEBUG(1, (sf, CE_CONT,
2403             "\sf_state callback, sf%d: "
2404             "got FCAL_STATE_RESET during DDI_SUSPEND\n",
2405             ddi_get_instance(sf->sft_dip)));
2406     } else {
2407         sf->sft_state = SF_STATE_OFFLINE;
2408     }
2409
2410     /*
2411      * scan each possible target on the loop, looking for targets
2412      * that need callbacks ran
2413      */
2414     for (i = 0; i < sf_max_targets; i++) {
2415         target = sf->sft_targets[i];
2416         while (target != NULL) {
2417             if (!(target->sft_state & SF_TARGET_OFFLINE)) {
2418                 target->sft_state |= (SF_TARGET_BUSY
2419                     | SF_TARGET_MARK);
2420                 mutex_exit(&sf->sft_mutex);
2421                 /*
2422                  * run remove event callbacks for lun
2423                  *
2424                  * We have a nasty race condition here
2425                  * 'cause we're dropping this mutex to
2426                  * run the callback and expect the
2427                  * linked list to be the same.
2428                  */
2429                 (void) ndi_event_retrieve_cookie(
2430                     sf->sft_event_hdl, target->sft_dip,
2431                     FCAL_REMOVE_EVENT, &sf_remove_eid,
2432                     NDI_EVENT_NOPASS);
2433                 (void) ndi_event_run_callbacks(
2434                     sf->sft_event_hdl,

```

```

2435         target->sft_dip,
2436         sf_remove_eid, NULL);
2437         mutex_enter(&sf->sft_mutex);
2438     }
2439     target = target->sft_next_lun;
2440 }
2441 }
2442
2443 /*
2444 * scan for ELS commands that are in transport, not complete,
2445 * and have a valid timeout, building a private list
2446 */
2447 privp = sf->sft_els_list;
2448 while (privp != NULL) {
2449     fpkt = privp->fpkt;
2450     if ((fpkt->fcml_cmd_state & FCAL_CMD_IN_TRANSPORT) &&
2451         (!fpkt->fcml_cmd_state & FCAL_CMD_COMPLETE)) &&
2452         (privp->timeout != SF_INVALID_TIMEOUT)) {
2453         /*
2454          * cmd in transport && not complete &&
2455          * timeout valid
2456          *
2457          * move this entry from ELS input list to our
2458          * private list
2459          */
2460         tmpp2 = privp->next; /* save ptr to next */
2461
2462         /* push this on private list head */
2463         privp->next = head;
2464         head = privp;
2465
2466         /* remove this entry from input list */
2467         if (tmpp1 != NULL) {
2468             /*
2469              * remove this entry from somewhere in
2470              * the middle of the list
2471              */
2472             tmpp1->next = tmpp2;
2473             if (tmpp2 != NULL) {
2474                 tmpp2->prev = tmpp1;
2475             }
2476         } else {
2477             /*
2478              * remove this entry from the head
2479              * of the list
2480              */
2481             sf->sft_els_list = tmpp2;
2482             if (tmpp2 != NULL) {
2483                 tmpp2->prev = NULL;
2484             }
2485         }
2486         privp = tmpp2; /* skip to next entry */
2487     } else {
2488         tmpp1 = privp; /* save ptr to prev entry */
2489         privp = privp->next; /* skip to next entry */
2490     }
2491 }
2492
2493 mutex_exit(&sf->sft_mutex);
2494
2495 /*
2496 * foreach cmd in our list free the ELS packet associated
2497 * with it
2498 */
2499 privp = head;
2500

```

```

2501     while (privp != NULL) {
2502         fpkt = privp->fpkt;
2503         privp = privp->next;
2504         sf_els_free(fpkt);
2505     }
2506
2507     /*
2508     * scan for commands from each possible target
2509     */
2510     for (i = 0; i < sf_max_targets; i++) {
2511         target = sf->sf_targets[i];
2512         while (target != NULL) {
2513             /*
2514             * scan all active commands for this target,
2515             * looking for commands that have been issued,
2516             * are in transport, and are not yet complete
2517             * (so we can terminate them because of the
2518             * reset)
2519             */
2520             mutex_enter(&target->sft_pkt_mutex);
2521             cmd = target->sft_pkt_head;
2522             while (cmd != (struct sf_pkt *)&target->
2523                 sft_pkt_head) {
2524                 fpkt = cmd->cmd_fp_pkt;
2525                 mutex_enter(&cmd->cmd_abort_mutex);
2526                 if ((cmd->cmd_state ==
2527                     SF_STATE_ISSUED) &&
2528                     (fpkt->fcml_cmd_state &
2529                     FCML_CMD_IN_TRANSPORT) &&
2530                     (!(fpkt->fcml_cmd_state &
2531                     FCML_CMD_COMPLETE))) {
2532                     /* a command to be reset */
2533                     pkt = cmd->cmd_pkt;
2534                     pkt->pkt_reason = CMD_RESET;
2535                     pkt->pkt_statistics |=
2536                         STAT_BUS_RESET;
2537                     cmd->cmd_state = SF_STATE_IDLE;
2538                     mutex_exit(&cmd->
2539                         cmd_abort_mutex);
2540                     mutex_exit(&target->
2541                         sft_pkt_mutex);
2542                     if (pkt->pkt_comp != NULL) {
2543                         (*pkt->pkt_comp)(pkt);
2544                     }
2545                     mutex_enter(&target->
2546                         sft_pkt_mutex);
2547                     cmd = target->sft_pkt_head;
2548                 } else {
2549                     mutex_exit(&cmd->
2550                         cmd_abort_mutex);
2551                     /* get next command */
2552                     cmd = cmd->cmd_forw;
2553                 }
2554             }
2555             mutex_exit(&target->sft_pkt_mutex);
2556             target = target->sft_next_lun;
2557         }
2558     }
2559
2560     /*
2561     * get packet queue for this target, resetting all remaining
2562     * commands
2563     */
2564     mutex_enter(&sf->sf_mutex);
2565     cmd = sf->sf_pkt_head;
2566     sf->sf_pkt_head = NULL;

```

```

2567         mutex_exit(&sf->sf_mutex);
2568
2569         while (cmd != NULL) {
2570             pkt = cmd->cmd_pkt;
2571             cmd = cmd->cmd_next;
2572             pkt->pkt_reason = CMD_RESET;
2573             pkt->pkt_statistics |= STAT_BUS_RESET;
2574             if (pkt->pkt_comp != NULL) {
2575                 (*pkt->pkt_comp)(pkt);
2576             }
2577         }
2578         break;
2579     }
2580
2581     default:
2582         break;
2583     }
2584 }
2585
2586 /*
2587 * called to send a PLOGI (N_port login) ELS request to a destination ID,
2588 * returning TRUE upon success, else returning FALSE
2589 */
2590 static int
2591 sf_login(struct sf *sf, uchar_t els_code, uchar_t dest_id, uint_t arg1,
2592          int lip_cnt)
2593 {
2594     struct la_els_logi    *logi;
2595     struct sf_els_hdr     *privp;
2596
2597     if (sf_els_alloc(sf, dest_id, sizeof (struct sf_els_hdr),
2598                     sizeof (union sf_els_cmd), sizeof (union sf_els_rsp),
2599                     (caddr_t *)&privp, (caddr_t *)&logi) == NULL) {
2600         sf_log(sf, CE_WARN, "Cannot allocate PLOGI for target %x "
2601             "due to DVMA shortage.\n", sf_alpa_to_switch[dest_id]);
2602         return (FALSE);
2603     }
2604
2605     privp->lip_cnt = lip_cnt;
2606     if (els_code == LA_ELS_PLOGI) {
2607         bcopy((caddr_t)sf->sf_sochandle->fcml_loginparms,
2608             (caddr_t)&logi->common_service, sizeof (struct la_els_logi)
2609             - 4);
2610         bcopy((caddr_t)&sf->sf_sochandle->fcml_p_wwn,
2611             (caddr_t)&logi->nport_ww_name, sizeof (la_wwn_t));
2612         bcopy((caddr_t)&sf->sf_sochandle->fcml_n_wwn,
2613             (caddr_t)&logi->node_ww_name, sizeof (la_wwn_t));
2614         bzero((caddr_t)&logi->reserved, 16);
2615     } else if (els_code == LA_ELS_LOGO) {
2616         bcopy((caddr_t)&sf->sf_sochandle->fcml_p_wwn,
2617             (caddr_t)&((struct la_els_logo *)logi)->nport_ww_name, 8);
2618         ((struct la_els_logo *)logi)->reserved = 0;
2619         ((struct la_els_logo *)logi)->nport_id[0] = 0;
2620         ((struct la_els_logo *)logi)->nport_id[1] = 0;
2621         ((struct la_els_logo *)logi)->nport_id[2] = arg1;
2622     }
2623
2624     privp->els_code = els_code;
2625     logi->ls_code = els_code;
2626     logi->mbz[0] = 0;
2627     logi->mbz[1] = 0;
2628     logi->mbz[2] = 0;
2629
2630     privp->timeout = sf_watchdog_time + SF_ELS_TIMEOUT;

```

```

2633     return (sf_els_transport(sf, privp));
2634 }

2637 /*
2638  * send an ELS IU via the transport,
2639  * returning TRUE upon success, else returning FALSE
2640  */
2641 static int
2642 sf_els_transport(struct sf *sf, struct sf_els_hdr *privp)
2643 {
2644     struct fcpl_packet *fpkt = privp->fpkt;

2647     (void) ddi_dma_sync(privp->cmd_dma_handle, (off_t)0, (size_t)0,
2648         DDI_DMA_SYNC_FORDEV);
2649     privp->prev = NULL;
2650     mutex_enter(&sf->sf_mutex);
2651     privp->next = sf->sf_els_list;
2652     if (sf->sf_els_list != NULL) {
2653         sf->sf_els_list->prev = privp;
2654     }
2655     sf->sf_els_list = privp;
2656     mutex_exit(&sf->sf_mutex);

2658     /* call the transport to send a packet */
2659     if (soc_transport(sf->sf_sochandle, fpkt, FCAL_NOSLEEP,
2660         CQ_REQUEST_1) != FCAL_TRANSPORT_SUCCESS) {
2661         mutex_enter(&sf->sf_mutex);
2662         if (privp->prev != NULL) {
2663             privp->prev->next = privp->next;
2664         }
2665         if (privp->next != NULL) {
2666             privp->next->prev = privp->prev;
2667         }
2668         if (sf->sf_els_list == privp) {
2669             sf->sf_els_list = privp->next;
2670         }
2671         mutex_exit(&sf->sf_mutex);
2672         sf_els_free(fpkt);
2673         return (FALSE);          /* failure */
2674     }
2675     return (TRUE);             /* success */
2676 }

2679 /*
2680  * called as the pkt_comp routine for ELS FC packets
2681  */
2682 static void
2683 sf_els_callback(struct fcpl_packet *fpkt)
2684 {
2685     struct sf_els_hdr *privp = fpkt->fcpl_pkt_private;
2686     struct sf *sf = privp->sf;
2687     struct sf *tsf;
2688     int tgt_id;
2689     struct la_els_logi *ptr = (struct la_els_logi *)privp->rsp;
2690     struct la_els_adisc *adisc = (struct la_els_adisc *)ptr;
2691     struct sf_target *target;
2692     short ncmds;
2693     short free_pkt = TRUE;

2696     /*
2697     * we've received an ELS callback, i.e. an ELS packet has arrived
2698     */

```

```

2700     /* take the current packet off of the queue */
2701     mutex_enter(&sf->sf_mutex);
2702     if (privp->timeout == SF_INVALID_TIMEOUT) {
2703         mutex_exit(&sf->sf_mutex);
2704         return;
2705     }
2706     if (privp->prev != NULL) {
2707         privp->prev->next = privp->next;
2708     }
2709     if (privp->next != NULL) {
2710         privp->next->prev = privp->prev;
2711     }
2712     if (sf->sf_els_list == privp) {
2713         sf->sf_els_list = privp->next;
2714     }
2715     privp->prev = privp->next = NULL;
2716     mutex_exit(&sf->sf_mutex);

2718     /* get # pkts in this callback */
2719     ncmds = fpkt->fcpl_ncmds;
2720     ASSERT(ncmds >= 0);
2721     mutex_enter(&sf->sf_cmd_mutex);
2722     sf->sf_ncmds = ncmds;
2723     mutex_exit(&sf->sf_cmd_mutex);

2725     /* sync idea of memory */
2726     (void) ddi_dma_sync(privp->rsp_dma_handle, (off_t)0, (size_t)0,
2727         DDI_DMA_SYNC_FORKERNEL);

2729     /* was this an OK ACC msg ?? */
2730     if ((fpkt->fcpl_pkt_status == FCAL_STATUS_OK) &&
2731         (ptr->ls_code == LA_ELS_ACC)) {

2733         /*
2734         * this was an OK ACC pkt
2735         */

2737         switch (privp->els_code) {
2738         case LA_ELS_PLOGI:
2739             /*
2740             * was able to to an N_port login
2741             */
2742             SF_DEBUG(2, (sf, CE_CONT,
2743                 "!PLOGI to al_pa %x succeeded, wwn %x%x\n",
2744                 privp->dest_nport_id,
2745                 *((int *)&ptr->nport_ww_name.raw_wwn[0]),
2746                 *((int *)&ptr->nport_ww_name.raw_wwn[4])));
2747             /* try to do a process login */
2748             if (!sf_do_pqli(sf, privp, ptr)) {
2749                 free_pkt = FALSE;
2750                 goto fail;      /* PRLI failed */
2751             }
2752             break;
2753         case LA_ELS_PRLI:
2754             /*
2755             * was able to do a process login
2756             */
2757             SF_DEBUG(2, (sf, CE_CONT,
2758                 "!PRLI to al_pa %x succeeded\n",
2759                 privp->dest_nport_id));
2760             /* try to do address discovery */
2761             if (sf_do_adisc(sf, privp) != 1) {
2762                 free_pkt = FALSE;
2763                 goto fail;      /* ADISC failed */
2764             }

```



```

2765         break;
2766     case LA_ELS_ADISC:
2767         /*
2768          * found a target via ADISC
2769          */
2771
2772         SF_DEBUG(2, (sf, CE_CONT,
2773                    "!ADISC to al_pa %x succeeded\n",
2774                    privp->dest_nport_id));
2775
2776         /* create the target info */
2777         if ((target = sf_create_target(sf, privp,
2778                                       sf_alpa_to_switch[(uchar_t)adisc->hard_address],
2779                                       (int64_t)0))
2780             == NULL) {
2781             goto fail; /* can't create target */
2782         }
2783
2784         /*
2785          * ensure address discovered matches what we thought
2786          * it would be
2787          */
2788         if ((uchar_t)adisc->hard_address !=
2789             privp->dest_nport_id) {
2790             sf_log(sf, CE_WARN,
2791                  "target 0x%x, AL-PA 0x%x and "
2792                  "hard address 0x%x don't match\n",
2793                  sf_alpa_to_switch[
2794                      (uchar_t)privp->dest_nport_id],
2795                  privp->dest_nport_id,
2796                  (uchar_t)adisc->hard_address);
2797             mutex_enter(&sf->sf_mutex);
2798             sf_offline_target(sf, target);
2799             mutex_exit(&sf->sf_mutex);
2800             goto fail; /* addr doesn't match */
2801         }
2802         /*
2803          * get inquiry data from the target
2804          */
2805         if (!sf_do_reportlun(sf, privp, target)) {
2806             mutex_enter(&sf->sf_mutex);
2807             sf_offline_target(sf, target);
2808             mutex_exit(&sf->sf_mutex);
2809             free_pkt = FALSE;
2810             goto fail; /* inquiry failed */
2811         }
2812         break;
2813     default:
2814         SF_DEBUG(2, (sf, CE_CONT,
2815                    "!ELS %x to al_pa %x succeeded\n",
2816                    privp->els_code, privp->dest_nport_id));
2817         sf_els_free(fpkt);
2818         break;
2819     }
2820 } else {
2821
2822     /*
2823     * oh oh -- this was not an OK ACC packet
2824     */
2825
2826     /* get target ID from dest loop address */
2827     tgt_id = sf_alpa_to_switch[(uchar_t)privp->dest_nport_id];
2828
2829     /* keep track of failures */
2830     sf->sf_stats.tstats[tgt_id].els_failures++;

```

```

2831         if (++(privp->retries) < sf_els_retries &&
2832             fpkt->fcal_pkt_status != FCAL_STATUS_OPEN_FAIL) {
2833             if (fpkt->fcal_pkt_status ==
2834                 FCAL_STATUS_MAX_XCHG_EXCEEDED) {
2835                 tsf = sf->sf_sibling;
2836                 if (tsf != NULL) {
2837                     mutex_enter(&tsf->sf_cmd_mutex);
2838                     tsf->sf_flag = 1;
2839                     tsf->sf_throttle = SF_DECR_DELTA;
2840                     mutex_exit(&tsf->sf_cmd_mutex);
2841                 }
2842             }
2843             privp->timeout = sf_watchdog_time + SF_ELS_TIMEOUT;
2844             privp->prev = NULL;
2845
2846             mutex_enter(&sf->sf_mutex);
2847
2848             if (privp->lip_cnt == sf->sf_lip_cnt) {
2849                 SF_DEBUG(1, (sf, CE_WARN,
2850                            "!ELS %x to al_pa %x failed, retrying",
2851                            privp->els_code, privp->dest_nport_id));
2852                 privp->next = sf->sf_els_list;
2853                 if (sf->sf_els_list != NULL) {
2854                     sf->sf_els_list->prev = privp;
2855                 }
2856                 sf->sf_els_list = privp;
2857
2858                 mutex_exit(&sf->sf_mutex);
2859                 /* device busy? wait a bit ... */
2860                 if (fpkt->fcal_pkt_status ==
2861                     FCAL_STATUS_MAX_XCHG_EXCEEDED) {
2862                     privp->delayed_retry = 1;
2863                     return;
2864                 }
2865                 /* call the transport to send a pkt */
2866                 if (soc_transport(sf->sf_sochandle, fpkt,
2867                                 FCAL_NOSLEEP, CQ_REQUEST_1) !=
2868                     FCAL_TRANSPORT_SUCCESS) {
2869                     mutex_enter(&sf->sf_mutex);
2870                     if (privp->prev != NULL) {
2871                         privp->prev->next =
2872                             privp->next;
2873                     }
2874                     if (privp->next != NULL) {
2875                         privp->next->prev =
2876                             privp->prev;
2877                     }
2878                     if (sf->sf_els_list == privp) {
2879                         sf->sf_els_list = privp->next;
2880                     }
2881                     mutex_exit(&sf->sf_mutex);
2882                     goto fail;
2883                 } else
2884                     return;
2885             } else {
2886                 mutex_exit(&sf->sf_mutex);
2887                 goto fail;
2888             }
2889         } else {
2890             #ifdef DEBUG
2891             if (fpkt->fcal_pkt_status != 0x36 || sfdebug > 4) {
2892                 SF_DEBUG(2, (sf, CE_NOTE, "ELS %x to al_pa %x failed",
2893                             privp->els_code, privp->dest_nport_id));
2894             }
2895             if (fpkt->fcal_pkt_status == FCAL_STATUS_OK) {
2896                 SF_DEBUG(2, (sf, CE_NOTE,

```

```

2897         "els reply code = %x", ptr->ls_code));
2898     if (ptr->ls_code == LA_ELS_RJT)
2899         SF_DEBUG(1, (sf, CE_CONT,
2900             "LS_RJT reason = %x\n",
2901             *((uint_t *)ptr) + 1));
2902     } else
2903         SF_DEBUG(2, (sf, CE_NOTE,
2904             "fc packet status = %x",
2905             fpkt->fcal_pkt_status));
2906     }
2907 #endif
2908     goto fail;
2909 }
2910 }
2911 return; /* success */
2912 fail:
2913 mutex_enter(&sf->sf_mutex);
2914 if (sf->sf_lip_cnt == privp->lip_cnt) {
2915     sf->sf_device_count--;
2916     ASSERT(sf->sf_device_count >= 0);
2917     if (sf->sf_device_count == 0) {
2918         sf_finish_init(sf, privp->lip_cnt);
2919     }
2920 }
2921 mutex_exit(&sf->sf_mutex);
2922 if (free_pkt) {
2923     sf_els_free(fpkt);
2924 }
2925 }

2928 /*
2929  * send a PRLI (process login) ELS IU via the transport,
2930  * returning TRUE upon success, else returning FALSE
2931  */
2932 static int
2933 sf_do_prli(struct sf *sf, struct sf_els_hdr *privp, struct la_els_logi *ptr)
2934 {
2935     struct la_els_prli *prli = (struct la_els_prli *)privp->cmd;
2936     struct fcp_prli *fprli;
2937     struct fcal_packet *fpkt = privp->fpkt;

2940     fpkt->fcal_socal_request.sr_dataseg[0].fc_count =
2941         sizeof(struct la_els_prli);
2942     privp->els_code = LA_ELS_PRLI;
2943     fprli = (struct fcp_prli *)prli->service_params;
2944     prli->ls_code = LA_ELS_PRLI;
2945     prli->page_length = 0x10;
2946     prli->payload_length = sizeof(struct la_els_prli);
2947     fprli->type = 0x08; /* no define here? */
2948     fprli->resvdl = 0;
2949     fprli->orig_process_assoc_valid = 0;
2950     fprli->resp_process_assoc_valid = 0;
2951     fprli->establish_image_pair = 1;
2952     fprli->resvdl2 = 0;
2953     fprli->resvdl3 = 0;
2954     fprli->data_overlay_allowed = 0;
2955     fprli->initiator_fn = 1;
2956     fprli->target_fn = 0;
2957     fprli->cmd_data_mixed = 0;
2958     fprli->data_resp_mixed = 0;
2959     fprli->read_xfer_rdy_disabled = 1;
2960     fprli->write_xfer_rdy_disabled = 0;

2962     bcopy((caddr_t)&ptr->nport_wv_name, (caddr_t)&privp->port_wvn,

```

```

2963         sizeof(privp->port_wvn));
2964     bcopy((caddr_t)&ptr->node_wv_name, (caddr_t)&privp->node_wvn,
2965         sizeof(privp->node_wvn));

2967     privp->timeout = sf_watchdog_time + SF_ELS_TIMEOUT;
2968     return (sf_els_transport(sf, privp));
2969 }

2972 /*
2973  * send an ADISC (address discovery) ELS IU via the transport,
2974  * returning TRUE upon success, else returning FALSE
2975  */
2976 static int
2977 sf_do_adisc(struct sf *sf, struct sf_els_hdr *privp)
2978 {
2979     struct la_els_adisc *adisc = (struct la_els_adisc *)privp->cmd;
2980     struct fcal_packet *fpkt = privp->fpkt;

2982     privp->els_code = LA_ELS_ADISC;
2983     adisc->ls_code = LA_ELS_ADISC;
2984     adisc->mbz[0] = 0;
2985     adisc->mbz[1] = 0;
2986     adisc->mbz[2] = 0;
2987     adisc->hard_address = 0; /* ??? */
2988     fpkt->fcal_socal_request.sr_dataseg[0].fc_count =
2989         sizeof(struct la_els_adisc);
2990     bcopy((caddr_t)&sf->sf_sochandle->fcal_p_wvn,
2991         (caddr_t)&adisc->port_wvn, sizeof(adisc->port_wvn));
2992     bcopy((caddr_t)&sf->sf_sochandle->fcal_n_wvn,
2993         (caddr_t)&adisc->node_wvn, sizeof(adisc->node_wvn));
2994     adisc->nport_id = sf->sf_al_pa;

2996     privp->timeout = sf_watchdog_time + SF_ELS_TIMEOUT;
2997     return (sf_els_transport(sf, privp));
2998 }

3001 static struct fcal_packet *
3002 sf_els_alloc(struct sf *sf, uchar_t dest_id, int priv_size, int cmd_size,
3003     int rsp_size, caddr_t *rprivp, caddr_t *cmd_buf)
3004 {
3005     struct fcal_packet *fpkt;
3006     ddi_dma_cookie_t pcookie;
3007     ddi_dma_cookie_t rcookie;
3008     struct sf_els_hdr *privp;
3009     ddi_dma_handle_t cmd_dma_handle = NULL;
3010     ddi_dma_handle_t rsp_dma_handle = NULL;
3011     ddi_acc_handle_t cmd_acc_handle = NULL;
3012     ddi_acc_handle_t rsp_acc_handle = NULL;
3013     size_t real_size;
3014     uint_t ccount;
3015     fc_frame_header_t *hp;
3016     int cmd_bound = FALSE, rsp_bound = FALSE;
3017     caddr_t cmd = NULL;
3018     caddr_t rsp = NULL;

3020     if ((fpkt = (struct fcal_packet *)kmem_zalloc(
3021         sizeof(struct fcal_packet), KM_NOSLEEP)) == NULL) {
3022         SF_DEBUG(1, (sf, CE_WARN,
3023             "Could not allocate fcal_packet for ELS\n"));
3024     }
3025     return (NULL);

3027     if ((privp = (struct sf_els_hdr *)kmem_zalloc(priv_size,
3028         KM_NOSLEEP)) == NULL) {

```

```

3029         SF_DEBUG(1, (sf, CE_WARN,
3030             "Could not allocate sf_els_hdr for ELS\n"));
3031         goto fail;
3032     }

3034     privp->size = priv_size;
3035     fpkt->fcal_pkt_private = (caddr_t)privp;

3037     if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
3038         DDI_DMA_DONTWAIT, NULL, &cmd_dma_handle) != DDI_SUCCESS) {
3039         SF_DEBUG(1, (sf, CE_WARN,
3040             "Could not allocate DMA handle for ELS\n"));
3041         goto fail;
3042     }

3044     if (ddi_dma_mem_alloc(cmd_dma_handle, cmd_size,
3045         sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
3046         DDI_DMA_DONTWAIT, NULL, &cmd,
3047         &real_size, &cmd_acc_handle) != DDI_SUCCESS) {
3048         SF_DEBUG(1, (sf, CE_WARN,
3049             "Could not allocate DMA memory for ELS\n"));
3050         goto fail;
3051     }

3053     if (real_size < cmd_size) {
3054         SF_DEBUG(1, (sf, CE_WARN,
3055             "DMA memory too small for ELS\n"));
3056         goto fail;
3057     }

3059     if (ddi_dma_addr_bind_handle(cmd_dma_handle, NULL,
3060         cmd, real_size, DDI_DMA_WRITE | DDI_DMA_CONSISTENT,
3061         DDI_DMA_DONTWAIT, NULL, &pcookie, &ccount) != DDI_DMA_MAPPED) {
3062         SF_DEBUG(1, (sf, CE_WARN,
3063             "Could not bind DMA memory for ELS\n"));
3064         goto fail;
3065     }
3066     cmd_bound = TRUE;

3068     if (ccount != 1) {
3069         SF_DEBUG(1, (sf, CE_WARN,
3070             "Wrong cookie count for ELS\n"));
3071         goto fail;
3072     }

3074     if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
3075         DDI_DMA_DONTWAIT, NULL, &rsp_dma_handle) != DDI_SUCCESS) {
3076         SF_DEBUG(1, (sf, CE_WARN,
3077             "Could not allocate DMA handle for ELS rsp\n"));
3078         goto fail;
3079     }
3080     if (ddi_dma_mem_alloc(rsp_dma_handle, rsp_size,
3081         sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
3082         DDI_DMA_DONTWAIT, NULL, &rsp,
3083         &real_size, &rsp_acc_handle) != DDI_SUCCESS) {
3084         SF_DEBUG(1, (sf, CE_WARN,
3085             "Could not allocate DMA memory for ELS rsp\n"));
3086         goto fail;
3087     }

3089     if (real_size < rsp_size) {
3090         SF_DEBUG(1, (sf, CE_WARN,
3091             "DMA memory too small for ELS rsp\n"));
3092         goto fail;
3093     }

```

```

3095     if (ddi_dma_addr_bind_handle(rsp_dma_handle, NULL,
3096         rsp, real_size, DDI_DMA_READ | DDI_DMA_CONSISTENT,
3097         DDI_DMA_DONTWAIT, NULL, &rcookie, &ccount) != DDI_DMA_MAPPED) {
3098         SF_DEBUG(1, (sf, CE_WARN,
3099             "Could not bind DMA memory for ELS rsp\n"));
3100         goto fail;
3101     }
3102     rsp_bound = TRUE;

3104     if (ccount != 1) {
3105         SF_DEBUG(1, (sf, CE_WARN,
3106             "Wrong cookie count for ELS rsp\n"));
3107         goto fail;
3108     }

3110     privp->cmd = cmd;
3111     privp->sf = sf;
3112     privp->cmd_dma_handle = cmd_dma_handle;
3113     privp->cmd_acc_handle = cmd_acc_handle;
3114     privp->rsp = rsp;
3115     privp->rsp_dma_handle = rsp_dma_handle;
3116     privp->rsp_acc_handle = rsp_acc_handle;
3117     privp->dest_nport_id = dest_id;
3118     privp->fpkt = fpkt;

3120     fpkt->fcal_pkt_cookie = sf->sf_socp;
3121     fpkt->fcal_pkt_comp = sf_els_callback;
3122     fpkt->fcal_magic = FCALP_MAGIC;
3123     fpkt->fcal_pkt_flags = 0;
3124     fpkt->fcal_socal_request.sr_soc_hdr.sh_flags =
3125         (ushort_t)(SOC_FC_HEADER | sf->sf_sochandle->fcal_portno);
3126     fpkt->fcal_socal_request.sr_soc_hdr.sh_class = 3;
3127     fpkt->fcal_socal_request.sr_soc_hdr.sh_seg_cnt = 2;
3128     fpkt->fcal_socal_request.sr_soc_hdr.sh_byte_cnt = cmd_size;
3129     fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_count = 1;
3130     fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_flags = 0;
3131     fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_seqno = 0;
3132     fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_type = CQ_TYPE_SIMPLE;
3133     fpkt->fcal_socal_request.sr_dataseg[0].fc_base = (uint32_t)
3134         pcookie.dmac_address;
3135     fpkt->fcal_socal_request.sr_dataseg[0].fc_count = cmd_size;
3136     fpkt->fcal_socal_request.sr_dataseg[1].fc_base = (uint32_t)
3137         rcookie.dmac_address;
3138     fpkt->fcal_socal_request.sr_dataseg[1].fc_count = rsp_size;

3140     /* Fill in the Fabric Channel Header */
3141     hp = &fpkt->fcal_socal_request.sr_fc_frame_hdr;
3142     hp->r_ctl = R_CTL_ELS_REQ;
3143     hp->d_id = dest_id;
3144     hp->s_id = sf->sf_al_pa;
3145     hp->type = TYPE_EXTENDED_LS;
3146     hp->reserved1 = 0;
3147     hp->f_ctl = F_CTL_SEQ_INITIATIVE | F_CTL_FIRST_SEQ;
3148     hp->seq_id = 0;
3149     hp->df_ctl = 0;
3150     hp->seq_cnt = 0;
3151     hp->ox_id = 0xffff;
3152     hp->rx_id = 0xffff;
3153     hp->ro = 0;

3155     *rprivp = (caddr_t)privp;
3156     *cmd_buf = cmd;
3157     return (fpkt);

3159 fail:
3160     if (cmd_dma_handle != NULL) {

```

```

3161         if (cmd_bound) {
3162             (void) ddi_dma_unbind_handle(cmd_dma_handle);
3163         }
3164         ddi_dma_free_handle(&cmd_dma_handle);
3165         privp->cmd_dma_handle = NULL;
3166     }
3167     if (rsp_dma_handle != NULL) {
3168         if (rsp_bound) {
3169             (void) ddi_dma_unbind_handle(rsp_dma_handle);
3170         }
3171         ddi_dma_free_handle(&rsp_dma_handle);
3172         privp->rsp_dma_handle = NULL;
3173     }
3174     sf_els_free(fpkt);
3175     return (NULL);
3176 }

3179 static void
3180 sf_els_free(struct fcml_packet *fpkt)
3181 {
3182     struct sf_els_hdr     *privp = fpkt->fcml_pkt_private;

3184     if (privp != NULL) {
3185         if (privp->cmd_dma_handle != NULL) {
3186             (void) ddi_dma_unbind_handle(privp->cmd_dma_handle);
3187             ddi_dma_free_handle(&privp->cmd_dma_handle);
3188         }
3189         if (privp->cmd != NULL) {
3190             ddi_dma_mem_free(&privp->cmd_acc_handle);
3191         }

3193         if (privp->rsp_dma_handle != NULL) {
3194             (void) ddi_dma_unbind_handle(privp->rsp_dma_handle);
3195             ddi_dma_free_handle(&privp->rsp_dma_handle);
3196         }

3198         if (privp->rsp != NULL) {
3199             ddi_dma_mem_free(&privp->rsp_acc_handle);
3200         }
3201         if (privp->data_dma_handle) {
3202             (void) ddi_dma_unbind_handle(privp->data_dma_handle);
3203             ddi_dma_free_handle(&privp->data_dma_handle);
3204         }
3205         if (privp->data_buf) {
3206             ddi_dma_mem_free(&privp->data_acc_handle);
3207         }
3208         kmem_free(privp, privp->size);
3209     }
3210     kmem_free(fpkt, sizeof (struct fcml_packet));
3211 }

3214 static struct sf_target *
3215 sf_create_target(struct sf *sf, struct sf_els_hdr *privp, int tnum, int64_t lun)
3216 {
3217     struct sf_target *target, *ntarget, *otarget, *ptarget;
3218     int hash;
3219 #ifdef RAID_LUNS
3220     int64_t orig_lun = lun;

3222     /* XXXX Work around SCSI limitations. */
3223     lun = *((short *)&lun);
3224 #endif
3225     ntarget = kmem_zalloc(sizeof (struct sf_target), KM_NOSLEEP);
3226     mutex_enter(&sf->sf_mutex);

```

```

3227     if (sf->sf_lip_cnt != privp->lip_cnt) {
3228         mutex_exit(&sf->sf_mutex);
3229         if (ntarget != NULL)
3230             kmem_free(ntarget, sizeof (struct sf_target));
3231         return (NULL);
3232     }

3234     target = sf_lookup_target(sf, privp->port_wwn, lun);
3235     if (lun != 0) {
3236         /*
3237          * Since LUNs != 0 are queued up after LUN == 0, find LUN == 0
3238          * and enqueue the new LUN.
3239          */
3240         if ((ptarget = sf_lookup_target(sf, privp->port_wwn,
3241             (int64_t)0)) == NULL) {
3242             /*
3243              * Yeep -- no LUN 0?
3244              */
3245             mutex_exit(&sf->sf_mutex);
3246             sf_log(sf, CE_WARN, "target 0x%x "
3247                 "lun %" PRIx64 ": No LUN 0\n", tnum, lun);
3248             if (ntarget != NULL)
3249                 kmem_free(ntarget, sizeof (struct sf_target));
3250             return (NULL);
3251         }
3252         mutex_enter(&ptarget->sft_mutex);
3253         if (target != NULL && ptarget->sft_lip_cnt == sf->sf_lip_cnt &&
3254             ptarget->sft_state & SF_TARGET_OFFLINE) {
3255             /* LUN 0 already finished, duplicate its state */
3256             mutex_exit(&ptarget->sft_mutex);
3257             sf_offline_target(sf, target);
3258             mutex_exit(&sf->sf_mutex);
3259             if (ntarget != NULL)
3260                 kmem_free(ntarget, sizeof (struct sf_target));
3261             return (target);
3262         } else if (target != NULL) {
3263             /*
3264              * LUN 0 online or not examined yet.
3265              * Try to bring the LUN back online
3266              */
3267             mutex_exit(&ptarget->sft_mutex);
3268             mutex_enter(&target->sft_mutex);
3269             target->sft_lip_cnt = privp->lip_cnt;
3270             target->sft_state |= SF_TARGET_BUSY;
3271             target->sft_state &= ~(SF_TARGET_OFFLINE |
3272                 SF_TARGET_MARK);
3273             target->sft_al_pa = (uchar_t)privp->dest_nport_id;
3274             target->sft_hard_address = sf_switch_to_alpa[tnum];
3275             mutex_exit(&target->sft_mutex);
3276             mutex_exit(&sf->sf_mutex);
3277             if (ntarget != NULL)
3278                 kmem_free(ntarget, sizeof (struct sf_target));
3279             return (target);
3280         }
3281         mutex_exit(&ptarget->sft_mutex);
3282         if (ntarget == NULL) {
3283             mutex_exit(&sf->sf_mutex);
3284             return (NULL);
3285         }
3286         /* Initialize new target structure */
3287         bcopy((caddr_t)&privp->node_wwn,
3288             (caddr_t)&ntarget->sft_node_wwn, sizeof (privp->node_wwn));
3289         bcopy((caddr_t)&privp->port_wwn,
3290             (caddr_t)&ntarget->sft_port_wwn, sizeof (privp->port_wwn));
3291         ntarget->sft_lun.l = lun;
3292 #ifdef RAID_LUNS

```

```

3293     ntarget->sft_lun.l = orig_lun;
3294     ntarget->sft_raid_lun = (uint_t)lun;
3295 #endif
3296     mutex_init(&ntarget->sft_mutex, NULL, MUTEX_DRIVER, NULL);
3297     mutex_init(&ntarget->sft_pkt_mutex, NULL, MUTEX_DRIVER, NULL);
3298     /* Don't let anyone use this till we finishup init. */
3299     mutex_enter(&ntarget->sft_mutex);
3300     mutex_enter(&ntarget->sft_pkt_mutex);
3301
3302     hash = SF_HASH(privp->port_wnn, lun);
3303     ntarget->sft_next = sf->sf_wnn_lists[hash];
3304     sf->sf_wnn_lists[hash] = ntarget;
3305
3306     ntarget->sft_lip_cnt = privp->lip_cnt;
3307     ntarget->sft_al_pa = (uchar_t)privp->dest_nport_id;
3308     ntarget->sft_hard_address = sf_switch_to_alpa[tnum];
3309     ntarget->sft_device_type = DTYPE_UNKNOWN;
3310     ntarget->sft_state = SF_TARGET_BUSY;
3311     ntarget->sft_pkt_head = (struct sf_pkt *)&ntarget->
3312     sft_pkt_head;
3313     ntarget->sft_pkt_tail = (struct sf_pkt *)&ntarget->
3314     sft_pkt_head;
3315
3316     mutex_enter(&ntarget->sft_mutex);
3317     /* Traverse the list looking for this target */
3318     for (target = ptarget; target->sft_next_lun;
3319         target = target->sft_next_lun) {
3320         otarget = target->sft_next_lun;
3321     }
3322     ntarget->sft_next_lun = target->sft_next_lun;
3323     target->sft_next_lun = ntarget;
3324     mutex_exit(&ntarget->sft_mutex);
3325     mutex_exit(&ntarget->sft_pkt_mutex);
3326     mutex_exit(&ntarget->sft_mutex);
3327     mutex_exit(&sf->sf_mutex);
3328     return (ntarget);
3329
3330 }
3331 if (target != NULL && target->sft_lip_cnt == sf->sf_lip_cnt) {
3332     /* It's been touched this LIP -- duplicate WNNs */
3333     sf_offline_target(sf, target); /* And all the baby targets */
3334     mutex_exit(&sf->sf_mutex);
3335     sf_log(sf, CE_WARN, "target 0x%x, duplicate port wwns\n",
3336         tnum);
3337     if (ntarget != NULL) {
3338         kmem_free(ntarget, sizeof (struct sf_target));
3339     }
3340     return (NULL);
3341 }
3342
3343 if ((otarget = sf->sf_targets[tnum]) != NULL) {
3344     /* Someone else is in our slot */
3345     mutex_enter(&otarget->sft_mutex);
3346     if (otarget->sft_lip_cnt == sf->sf_lip_cnt) {
3347         mutex_exit(&otarget->sft_mutex);
3348         sf_offline_target(sf, otarget);
3349         if (target != NULL)
3350             sf_offline_target(sf, target);
3351         mutex_exit(&sf->sf_mutex);
3352         sf_log(sf, CE_WARN,
3353             "target 0x%x, duplicate switch settings\n", tnum);
3354         if (ntarget != NULL)
3355             kmem_free(ntarget, sizeof (struct sf_target));
3356         return (NULL);
3357     }
3358     mutex_exit(&otarget->sft_mutex);

```

```

3359         if (bcmp((caddr_t)&privp->port_wnn, (caddr_t)&otarget->
3360             sft_port_wnn, sizeof (privp->port_wnn))) {
3361             sf_offline_target(sf, otarget);
3362             mutex_exit(&sf->sf_mutex);
3363             sf_log(sf, CE_WARN, "wnn changed on target 0x%x\n",
3364                 tnum);
3365             bzero((caddr_t)&sf->sf_stats.tstats[tnum],
3366                 sizeof (struct sf_target_stats));
3367             mutex_enter(&sf->sf_mutex);
3368         }
3369     }
3370
3371     sf->sf_targets[tnum] = target;
3372     if ((target = sf->sf_targets[tnum]) == NULL) {
3373         if (ntarget == NULL) {
3374             mutex_exit(&sf->sf_mutex);
3375             return (NULL);
3376         }
3377         bcopy((caddr_t)&privp->node_wnn,
3378             (caddr_t)&ntarget->sft_node_wnn, sizeof (privp->node_wnn));
3379         bcopy((caddr_t)&privp->port_wnn,
3380             (caddr_t)&ntarget->sft_port_wnn, sizeof (privp->port_wnn));
3381         ntarget->sft_lun.l = lun;
3382 #ifdef RAID_LUNS
3383         ntarget->sft_lun.l = orig_lun;
3384         ntarget->sft_raid_lun = (uint_t)lun;
3385 #endif
3386         mutex_init(&ntarget->sft_mutex, NULL, MUTEX_DRIVER, NULL);
3387         mutex_init(&ntarget->sft_pkt_mutex, NULL, MUTEX_DRIVER, NULL);
3388         mutex_enter(&ntarget->sft_mutex);
3389         mutex_enter(&ntarget->sft_pkt_mutex);
3390         hash = SF_HASH(privp->port_wnn, lun); /* lun 0 */
3391         ntarget->sft_next = sf->sf_wnn_lists[hash];
3392         sf->sf_wnn_lists[hash] = ntarget;
3393
3394         target = ntarget;
3395         target->sft_lip_cnt = privp->lip_cnt;
3396         target->sft_al_pa = (uchar_t)privp->dest_nport_id;
3397         target->sft_hard_address = sf_switch_to_alpa[tnum];
3398         target->sft_device_type = DTYPE_UNKNOWN;
3399         target->sft_state = SF_TARGET_BUSY;
3400         target->sft_pkt_head = (struct sf_pkt *)&target->
3401         sft_pkt_head;
3402         target->sft_pkt_tail = (struct sf_pkt *)&target->
3403         sft_pkt_head;
3404         sf->sf_targets[tnum] = target;
3405         mutex_exit(&ntarget->sft_mutex);
3406         mutex_exit(&ntarget->sft_pkt_mutex);
3407         mutex_exit(&sf->sf_mutex);
3408     } else {
3409         mutex_enter(&target->sft_mutex);
3410         target->sft_lip_cnt = privp->lip_cnt;
3411         target->sft_state |= SF_TARGET_BUSY;
3412         target->sft_state &= ~(SF_TARGET_OFFLINE|SF_TARGET_MARK);
3413         target->sft_al_pa = (uchar_t)privp->dest_nport_id;
3414         target->sft_hard_address = sf_switch_to_alpa[tnum];
3415         mutex_exit(&target->sft_mutex);
3416         mutex_exit(&sf->sf_mutex);
3417         if (ntarget != NULL)
3418             kmem_free(ntarget, sizeof (struct sf_target));
3419     }
3420     return (target);
3421 }
3422
3424 /*

```

```

3425 * find the target for a given sf instance
3426 */
3427 /* ARGSUSED */
3428 static struct sf_target *
3429 #ifdef RAID_LUNS
3430 sf_lookup_target(struct sf *sf, uchar_t *wwn, int lun)
3431 #else
3432 sf_lookup_target(struct sf *sf, uchar_t *wwn, int64_t lun)
3433 #endif
3434 {
3435     int hash;
3436     struct sf_target *target;
3437
3438     ASSERT(mutex_owned(&sf->sf_mutex));
3439     hash = SF_HASH(wwn, lun);
3440
3441     target = sf->sf_wwn_lists[hash];
3442     while (target != NULL) {
3443
3444 #ifndef RAID_LUNS
3445         if (bcmp((caddr_t)wwn, (caddr_t)&target->sft_port_wwn,
3446             sizeof (target->sft_port_wwn)) == 0 &&
3447             target->sft_lun.l == lun)
3448             break;
3449 #else
3450         if (bcmp((caddr_t)wwn, (caddr_t)&target->sft_port_wwn,
3451             sizeof (target->sft_port_wwn)) == 0 &&
3452             target->sft_raid_lun == lun)
3453             break;
3454 #endif
3455         target = target->sft_next;
3456     }
3457
3458     return (target);
3459 }
3460
3461 /*
3462 * Send out a REPORT_LUNS command.
3463 */
3464 static int
3465 sf_do_reportlun(struct sf *sf, struct sf_els_hdr *privp,
3466     struct sf_target *target)
3467 {
3468     struct fcal_packet *fpkt = privp->fpkt;
3469     ddi_dma_cookie_t pcookie;
3470     ddi_dma_handle_t lun_dma_handle = NULL;
3471     ddi_acc_handle_t lun_acc_handle;
3472     uint_t ccount;
3473     size_t real_size;
3474     caddr_t lun_buf = NULL;
3475     int handle_bound = 0;
3476     fc_frame_header_t *hp = &fpkt->fcal_socal_request.sr_fc_frame_hdr;
3477     struct fcp_cmd *reportlun = (struct fcp_cmd *)privp->cmd;
3478     char *msg = "Transport";
3479
3480     if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
3481         DDI_DMA_DONTWAIT, NULL, &lun_dma_handle) != DDI_SUCCESS) {
3482         msg = "ddi_dma_alloc_handle()";
3483         goto fail;
3484     }
3485
3486     if (ddi_dma_mem_alloc(lun_dma_handle, REPORT_LUNS_SIZE,
3487         sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
3488         DDI_DMA_DONTWAIT, NULL, &lun_buf,
3489         &real_size, &lun_acc_handle) != DDI_SUCCESS) {

```

```

3491         msg = "ddi_dma_mem_alloc()";
3492         goto fail;
3493     }
3494
3495     if (real_size < REPORT_LUNS_SIZE) {
3496         msg = "DMA mem < REPORT_LUNS_SIZE";
3497         goto fail;
3498     }
3499
3500     if (ddi_dma_addr_bind_handle(lun_dma_handle, NULL,
3501         lun_buf, real_size, DDI_DMA_READ |
3502         DDI_DMA_CONSISTENT, DDI_DMA_DONTWAIT,
3503         NULL, &pcookie, &ccount) != DDI_DMA_MAPPED) {
3504         msg = "ddi_dma_addr_bind_handle()";
3505         goto fail;
3506     }
3507     handle_bound = 1;
3508
3509     if (ccount != 1) {
3510         msg = "ccount != 1";
3511         goto fail;
3512     }
3513     privp->els_code = 0;
3514     privp->target = target;
3515     privp->data_dma_handle = lun_dma_handle;
3516     privp->data_acc_handle = lun_acc_handle;
3517     privp->data_buf = lun_buf;
3518
3519     fpkt->fcal_pkt_comp = sf_reportlun_callback;
3520     fpkt->fcal_socal_request.sr_soc_hdr.sh_seg_cnt = 3;
3521     fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_type = CQ_TYPE_IO_READ;
3522     fpkt->fcal_socal_request.sr_dataseg[0].fc_count =
3523         sizeof (struct fcp_cmd);
3524     fpkt->fcal_socal_request.sr_dataseg[2].fc_base =
3525         (uint32_t)pcookie.dmac_address;
3526     fpkt->fcal_socal_request.sr_dataseg[2].fc_count = pcookie.dmac_size;
3527     fpkt->fcal_socal_request.sr_soc_hdr.sh_byte_cnt = pcookie.dmac_size;
3528     hp->r_ctl = R_CTL_COMMAND;
3529     hp->ttype = TYPE_SCSI_FCP;
3530     bzero((caddr_t)reportlun, sizeof (struct fcp_cmd));
3531     ((union scsi_cdb *)reportlun->fcp_cdb)->scm5_cmd = SCMD_REPORT_LUNS;
3532     /* Now set the buffer size. If DDI gave us extra, that's O.K. */
3533     ((union scsi_cdb *)reportlun->fcp_cdb)->scm5_count0 =
3534         (real_size & 0x0fff);
3535     ((union scsi_cdb *)reportlun->fcp_cdb)->scm5_count1 =
3536         (real_size >> 8) & 0x0fff;
3537     ((union scsi_cdb *)reportlun->fcp_cdb)->scm5_count2 =
3538         (real_size >> 16) & 0x0fff;
3539     ((union scsi_cdb *)reportlun->fcp_cdb)->scm5_count3 =
3540         (real_size >> 24) & 0x0fff;
3541     reportlun->fcp_cntl.cntl_read_data = 1;
3542     reportlun->fcp_cntl.cntl_write_data = 0;
3543     reportlun->fcp_data_len = pcookie.dmac_size;
3544     reportlun->fcp_cntl.cntl_qtype = FCP_QTYPE_SIMPLE;
3545
3546     (void) ddi_dma_sync(lun_dma_handle, 0, 0, DDI_DMA_SYNC_FORDEV);
3547     /* We know he's there, so this should be fast */
3548     privp->timeout = sf_watchdog_time + SF_FCP_TIMEOUT;
3549     if (sf_els_transport(sf, privp) == 1)
3550         return (1);
3551
3552 fail:
3553     sf_log(sf, CE_WARN,
3554         "%s failure for REPORTLUN to target 0x%x\n",
3555         msg, sf_alpa_to_switch[privp->dest_nport_id]);
3556     sf_els_free(fpkt);

```

```

3557     if (lun_dma_handle != NULL) {
3558         if (handle_bound)
3559             (void) ddi_dma_unbind_handle(lun_dma_handle);
3560         ddi_dma_free_handle(&lun_dma_handle);
3561     }
3562     if (lun_buf != NULL) {
3563         ddi_dma_mem_free(&lun_acc_handle);
3564     }
3565     return (0);
3566 }

3568 /*
3569  * Handle the results of a REPORT LUNS command:
3570  *   Create additional targets if necessary
3571  *   Initiate INQUIRYs on all LUNs.
3572  */
3573 static void
3574 sf_reportlun_callback(struct fcal_packet *fpkt)
3575 {
3576     struct sf_els_hdr *privp = (struct sf_els_hdr *)fpkt->
3577         fcal_pkt_private;
3578     struct scsi_report_luns *ptr =
3579         (struct scsi_report_luns *)privp->data_buf;
3580     struct sf *sf = privp->sf;
3581     struct sf_target *target = privp->target;
3582     struct fcp_rsp *rsp = NULL;
3583     int delayed_retry = 0;
3584     int tid = sf_alpa_to_switch[target->sft_hard_address];
3585     int i, free_pkt = 1;
3586     short  ncmds;

3588     mutex_enter(&sf->sf_mutex);
3589     /* use as temporary state variable */
3590     if (privp->timeout == SF_INVALID_TIMEOUT) {
3591         mutex_exit(&sf->sf_mutex);
3592         return;
3593     }
3594     if (privp->prev)
3595         privp->prev->next = privp->next;
3596     if (privp->next)
3597         privp->next->prev = privp->prev;
3598     if (sf->sf_els_list == privp)
3599         sf->sf_els_list = privp->next;
3600     privp->prev = privp->next = NULL;
3601     mutex_exit(&sf->sf_mutex);
3602     ncmds = fpkt->fcal_ncmds;
3603     ASSERT(ncmds >= 0);
3604     mutex_enter(&sf->sf_cmd_mutex);
3605     sf->sf_ncmds = ncmds;
3606     mutex_exit(&sf->sf_cmd_mutex);

3608     if (fpkt->fcal_pkt_status == FCAL_STATUS_OK) {
3609         (void) ddi_dma_sync(privp->rsp_dma_handle, 0,
3610             0, DDI_DMA_SYNC_FORKERNEL);

3612         rsp = (struct fcp_rsp *)privp->rsp;
3613     }
3614     SF_DEBUG(1, (sf, CE_CONT,
3615         "!REPORTLUN to al_pa %x pkt status %x scsi status %x\n",
3616         privp->dest_nport_id,
3617         fpkt->fcal_pkt_status,
3618         rsp?rsp->fcp_u.fcp_status.scsi_status:0));

3620     /* See if target simply does not support REPORT LUNS. */
3621     if (rsp && rsp->fcp_u.fcp_status.scsi_status == STATUS_CHECK &&
3622         rsp->fcp_u.fcp_status.sense_len_set &&

```

```

3623         rsp->fcp_sense_len >=
3624         offsetof(struct scsi_extended_sense, es_qual_code)) {
3625         struct scsi_extended_sense *sense;
3626         sense = (struct scsi_extended_sense *)
3627             ((caddr_t)rsp + sizeof (struct fcp_rsp)
3628             + rsp->fcp_response_len);
3629         if (sense->es_key == KEY_ILLEGAL_REQUEST) {
3630             if (sense->es_add_code == 0x20) {
3631                 /* Fake LUN 0 */
3632                 SF_DEBUG(1, (sf, CE_CONT,
3633                     "!REPORTLUN Faking good "
3634                     "completion for alpa %x\n",
3635                     privp->dest_nport_id));
3636                 ptr->lun_list_len = FCP_LUN_SIZE;
3637                 ptr->lun[0] = 0;
3638                 rsp->fcp_u.fcp_status.scsi_status =
3639                     STATUS_GOOD;
3640             } else if (sense->es_add_code == 0x25) {
3641                 SF_DEBUG(1, (sf, CE_CONT,
3642                     "!REPORTLUN device alpa %x "
3643                     "key %x code %x\n",
3644                     privp->dest_nport_id,
3645                     sense->es_key, sense->es_add_code));
3646                 goto fail;
3647             }
3648         } else if (sense->es_key ==
3649             KEY_UNIT_ATTENTION &&
3650             sense->es_add_code == 0x29) {
3651             SF_DEBUG(1, (sf, CE_CONT,
3652                 "!REPORTLUN device alpa %x was reset\n",
3653                 privp->dest_nport_id));
3654         } else {
3655             SF_DEBUG(1, (sf, CE_CONT,
3656                 "!REPORTLUN device alpa %x "
3657                 "key %x code %x\n",
3658                 privp->dest_nport_id,
3659                 sense->es_key, sense->es_add_code));
3660             /* XXXXXX The following is here to handle broken targets -- remove it later */
3661             if (sf_reportlun_forever &&
3662                 sense->es_key == KEY_UNIT_ATTENTION)
3663                 goto retry;
3664             /* XXXXXX */
3665             if (sense->es_key == KEY_NOT_READY)
3666                 delayed_retry = 1;
3667         }
3668     }

3670     if (rsp && rsp->fcp_u.fcp_status.scsi_status == STATUS_GOOD) {
3671         struct fcp_rsp_info *bep;

3673         bep = (struct fcp_rsp_info *)&rsp->
3674             fcp_response_len + 1;
3675         if (!rsp->fcp_u.fcp_status.rsp_len_set ||
3676             bep->rsp_code == FCP_NO_FAILURE) {
3677             (void) ddi_dma_sync(privp->data_dma_handle,
3678                 0, 0, DDI_DMA_SYNC_FORKERNEL);

3680             /* Convert from #bytes to #ints */
3681             ptr->lun_list_len = ptr->lun_list_len >> 3;
3682             SF_DEBUG(2, (sf, CE_CONT,
3683                 "!REPORTLUN to al_pa %x succeeded: %d LUNs\n",
3684                 privp->dest_nport_id, ptr->lun_list_len));
3685             if (!ptr->lun_list_len) {
3686                 /* No LUNs? Ya gotta be kidding... */
3687                 sf_log(sf, CE_WARN,
3688                     "SCSI violation -- "

```

```

3689         "target 0x%x reports no LUNs\n",
3690         sf_alpa_to_switch[
3691         privp->dest_nport_id]);
3692         ptr->lun_list_len = 1;
3693         ptr->lun[0] = 0;
3694     }
3695
3696     mutex_enter(&sf->sf_mutex);
3697     if (sf->sf_lip_cnt == privp->lip_cnt) {
3698         sf->sf_device_count += ptr->lun_list_len - 1;
3699     }
3700
3701     mutex_exit(&sf->sf_mutex);
3702     for (i = 0; i < ptr->lun_list_len && privp->lip_cnt ==
3703         sf->sf_lip_cnt; i++) {
3704         struct sf_els_hdr *nprivp;
3705         struct fc_lun_packet *nfpkt;
3706
3707         /* LUN 0 is already in 'target' */
3708         if (ptr->lun[i] != 0) {
3709             target = sf_create_target(sf,
3710                 privp, tid, ptr->lun[i]);
3711         }
3712         nprivp = NULL;
3713         nfpkt = NULL;
3714         if (target) {
3715             nfpkt = sf_els_alloc(sf,
3716                 target->sft_al_pa,
3717                 sizeof (struct sf_els_hdr),
3718                 sizeof (union sf_els_cmd),
3719                 sizeof (union sf_els_rsp),
3720                 (caddr_t *)&nprivp,
3721                 (caddr_t *)&nfpkt);
3722             if (nprivp)
3723                 nprivp->lip_cnt =
3724                     privp->lip_cnt;
3725         }
3726         if (nfpkt && nprivp &&
3727             (sf_do_inquiry(sf, nprivp, target) ==
3728             0)) {
3729             mutex_enter(&sf->sf_mutex);
3730             if (sf->sf_lip_cnt == privp->
3731                 lip_cnt) {
3732                 sf->sf_device_count --;
3733             }
3734             sf_offline_target(sf, target);
3735             mutex_exit(&sf->sf_mutex);
3736         }
3737     }
3738     sf_els_free(fpkt);
3739     return;
3740 } else {
3741     SF_DEBUG(1, (sf, CE_CONT,
3742         "!REPORTLUN al_pa %x fcp failure, "
3743         "fcp_rsp_code %x scsi status %x\n",
3744         privp->dest_nport_id, bep->rsp_code,
3745         rsp ? rsp->fcp_u.fcp_status.scsi_status:0));
3746     goto fail;
3747 }
3748 }
3749 if (rsp && ((rsp->fcp_u.fcp_status.scsi_status == STATUS_BUSY) ||
3750     (rsp->fcp_u.fcp_status.scsi_status == STATUS_QFULL))) {
3751     delayed_retry = 1;
3752 }
3753
3754 if (++(privp->retries) < sf_els_retries ||

```

```

3755     (delayed_retry && privp->retries < SF_BSY_RETRIES)) {
3756     /* XXXXXX The following is here to handle broken targets -- remove it later */
3757     retry:
3758     /* XXXXXX */
3759     if (delayed_retry) {
3760         privp->retries--;
3761         privp->timeout = sf_watchdog_time + SF_BSY_TIMEOUT;
3762         privp->delayed_retry = 1;
3763     } else {
3764         privp->timeout = sf_watchdog_time + SF_FCP_TIMEOUT;
3765     }
3766
3767     privp->prev = NULL;
3768     mutex_enter(&sf->sf_mutex);
3769     if (privp->lip_cnt == sf->sf_lip_cnt) {
3770         if (!delayed_retry)
3771             SF_DEBUG(1, (sf, CE_WARN,
3772                 "!REPORTLUN to al_pa %x failed, retrying\n",
3773                 privp->dest_nport_id));
3774         privp->next = sf->sf_els_list;
3775         if (sf->sf_els_list != NULL)
3776             sf->sf_els_list->prev = privp;
3777         sf->sf_els_list = privp;
3778         mutex_exit(&sf->sf_mutex);
3779         if (!delayed_retry && soc_transport(sf->sf_sochandle,
3780             fpkt, FCAL_NOSLEEP, CQ_REQUEST_1) !=
3781             FCAL_TRANSPORT_SUCCESS) {
3782             mutex_enter(&sf->sf_mutex);
3783             if (privp->prev)
3784                 privp->prev->next = privp->next;
3785             if (privp->next)
3786                 privp->next->prev = privp->prev;
3787             if (sf->sf_els_list == privp)
3788                 sf->sf_els_list = privp->next;
3789             mutex_exit(&sf->sf_mutex);
3790             goto fail;
3791         } else
3792             return;
3793     } else {
3794         mutex_exit(&sf->sf_mutex);
3795     }
3796 } else {
3797 fail:
3798
3799     /* REPORT_LUN failed -- try inquiry */
3800     if (sf_do_inquiry(sf, privp, target) != 0) {
3801         return;
3802     } else {
3803         free_pkt = 0;
3804     }
3805     mutex_enter(&sf->sf_mutex);
3806     if (sf->sf_lip_cnt == privp->lip_cnt) {
3807         sf_log(sf, CE_WARN,
3808             "!REPORTLUN to target 0x%x failed\n",
3809             sf_alpa_to_switch[privp->dest_nport_id]);
3810         sf_offline_target(sf, target);
3811         sf->sf_device_count--;
3812         ASSERT(sf->sf_device_count >= 0);
3813         if (sf->sf_device_count == 0)
3814             sf_finish_init(sf, privp->lip_cnt);
3815     }
3816     mutex_exit(&sf->sf_mutex);
3817 }
3818 if (free_pkt) {
3819     sf_els_free(fpkt);
3820 }

```



```

3821 }

3823 static int
3824 sf_do_inquiry(struct sf *sf, struct sf_els_hdr *privp,
3825             struct sf_target *target)
3826 {
3827     struct fcal_packet      *fpkt = privp->fpkt;
3828     ddi_dma_cookie_t        pcookie;
3829     ddi_dma_handle_t        inq_dma_handle = NULL;
3830     ddi_acc_handle_t        inq_acc_handle;
3831     uint_t                  ccount;
3832     size_t                  real_size;
3833     caddr_t                 inq_buf = NULL;
3834     int                     handle_bound = FALSE;
3835     fc_frame_header_t *hp = &fpkt->fcal_socal_request.sr_fc_frame_hdr;
3836     struct fcp_cmd          *inq = (struct fcp_cmd *)privp->cmd;
3837     char                    *msg = "Transport";

    if (ddi_dma_alloc_handle(sf->sf_dip, sf->sf_sochandle->fcal_dmaattr,
3840     DDI_DMA_DONTWAIT, NULL, &inq_dma_handle) != DDI_SUCCESS) {
3841         msg = "ddi_dma_alloc_handle()";
3842         goto fail;
3843     }
3844 }

3846 if (ddi_dma_mem_alloc(inq_dma_handle, SUN_INQSIZE,
3847 sf->sf_sochandle->fcal_accattr, DDI_DMA_CONSISTENT,
3848 DDI_DMA_DONTWAIT, NULL, &inq_buf,
3849 &real_size, &inq_acc_handle) != DDI_SUCCESS) {
3850     msg = "ddi_dma_mem_alloc()";
3851     goto fail;
3852 }

3854 if (real_size < SUN_INQSIZE) {
3855     msg = "DMA mem < inquiry size";
3856     goto fail;
3857 }

3859 if (ddi_dma_addr_bind_handle(inq_dma_handle, NULL,
3860 inq_buf, real_size, DDI_DMA_READ | DDI_DMA_CONSISTENT,
3861 DDI_DMA_DONTWAIT, NULL, &pcookie, &ccount) != DDI_DMA_MAPPED) {
3862     msg = "ddi_dma_addr_bind_handle()";
3863     goto fail;
3864 }
3865 handle_bound = TRUE;

3867 if (ccount != 1) {
3868     msg = "ccount != 1";
3869     goto fail;
3870 }
3871 privp->els_code = 0; /* not an ELS command */
3872 privp->target = target;
3873 privp->data_dma_handle = inq_dma_handle;
3874 privp->data_acc_handle = inq_acc_handle;
3875 privp->data_buf = inq_buf;
3876 fpkt->fcal_pkt_comp = sf_inq_callback;
3877 fpkt->fcal_socal_request.sr_soc_hdr.sh_seg_cnt = 3;
3878 fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_type = CQ_TYPE_IO_READ;
3879 fpkt->fcal_socal_request.sr_dataseg[0].fc_count =
3880     sizeof (struct fcp_cmd);
3881 fpkt->fcal_socal_request.sr_dataseg[2].fc_base =
3882     (uint32_t)pcookie.dmac_address;
3883 fpkt->fcal_socal_request.sr_dataseg[2].fc_count = pcookie.dmac_size;
3884 fpkt->fcal_socal_request.sr_soc_hdr.sh_byte_cnt = pcookie.dmac_size;
3885 hp->r_ctl = R_CTL_COMMAND;
3886 hp->type = TYPE_SCSI_FCP;

```

```

3887     bzero((caddr_t)inq, sizeof (struct fcp_cmd));
3888     ((union scsi_cdb *)inq->fcp_cdb)->scm_cmd = SCMD_INQUIRY;
3889     ((union scsi_cdb *)inq->fcp_cdb)->g0_count0 = SUN_INQSIZE;
3890     bcopy((caddr_t)&target->sft_lun.b, (caddr_t)&inq->fcp_ent_addr,
3891         FCP_LUN_SIZE);
3892     inq->fcp_cntl.cntl_read_data = 1;
3893     inq->fcp_cntl.cntl_write_data = 0;
3894     inq->fcp_data_len = pcookie.dmac_size;
3895     inq->fcp_cntl.cntl_qtype = FCP_QTYPE_SIMPLE;

3897     (void) ddi_dma_sync(inq_dma_handle, (off_t)0, (size_t)0,
3898         DDI_DMA_SYNC_FORDEV);
3899     privp->timeout = sf_watchdog_time + SF_FCP_TIMEOUT;
3900     SF_DEBUG(5, (sf, CE_WARN,
3901         "!Sending INQUIRY to al_pa %x lun %" PRIx64 "\n",
3902         privp->dest_nport_id,
3903         SCSA_LUN(target)));
3904     return (sf_els_transport(sf, privp));

3906 fail:
3907     sf_log(sf, CE_WARN,
3908         "%s failure for INQUIRY to target 0x%x\n",
3909         msg, sf_alpa_to_switch[privp->dest_nport_id]);
3910     sf_els_free(fpkt);
3911     if (inq_dma_handle != NULL) {
3912         if (handle_bound) {
3913             (void) ddi_dma_unbind_handle(inq_dma_handle);
3914         }
3915         ddi_dma_free_handle(&inq_dma_handle);
3916     }
3917     if (inq_buf != NULL) {
3918         ddi_dma_mem_free(&inq_acc_handle);
3919     }
3920     return (FALSE);
3921 }

3924 /*
3925  * called as the pkt_comp routine for INQ packets
3926  */
3927 static void
3928 sf_inq_callback(struct fcal_packet *fpkt)
3929 {
3930     struct sf_els_hdr *privp = (struct sf_els_hdr *)fpkt->
3931         fcal_pkt_private;
3932     struct scsi_inquiry *prt = (struct scsi_inquiry *)privp->data_buf;
3933     struct sf *sf = privp->sf;
3934     struct sf *tsf;
3935     struct sf_target *target = privp->target;
3936     struct fcp_rsp *rsp;
3937     int delayed_retry = FALSE;
3938     short ncmds;

3941     mutex_enter(&sf->sf_mutex);
3942     /* use as temporary state variable */
3943     if (privp->timeout == SF_INVALID_TIMEOUT) {
3944         mutex_exit(&sf->sf_mutex);
3945         return;
3946     }
3947     if (privp->prev != NULL) {
3948         privp->prev->next = privp->next;
3949     }
3950     if (privp->next != NULL) {
3951         privp->next->prev = privp->prev;
3952     }

```

```

3953     if (sf->sf_els_list == privp) {
3954         sf->sf_els_list = privp->next;
3955     }
3956     privp->prev = privp->next = NULL;
3957     mutex_exit(&sf->sf_mutex);
3958     ncmds = fpkt->fcal_ncmds;
3959     ASSERT(ncmds >= 0);
3960     mutex_enter(&sf->sf_cmd_mutex);
3961     sf->sf_ncmds = ncmds;
3962     mutex_exit(&sf->sf_cmd_mutex);
3964     if (fpkt->fcal_pkt_status == FCAL_STATUS_OK) {
3966         (void) ddi_dma_sync(privp->rsp_dma_handle, (off_t)0,
3967             (size_t)0, DDI_DMA_SYNC_FORKERNEL);
3969         rsp = (struct fcp_rsp *)privp->rsp;
3970         SF_DEBUG(2, (sf, CE_CONT,
3971             "!INQUIRY to al_pa %x scsi status %x",
3972             privp->dest_nport_id, rsp->fcp_u.fcp_status.scsi_status));
3974         if ((rsp->fcp_u.fcp_status.scsi_status == STATUS_GOOD) &&
3975             !rsp->fcp_u.fcp_status.resid_over &&
3976             (!rsp->fcp_u.fcp_status.resid_under ||
3977             ((SUN_INQSIZE - rsp->fcp_resid) >= SUN_MIN_INQLEN))) {
3978             struct fcp_rsp_info *bep;
3980             bep = (struct fcp_rsp_info *)&rsp->
3981                 fcp_response_len + 1);
3983             if (!rsp->fcp_u.fcp_status.rsp_len_set ||
3984                 (bep->rsp_code == FCP_NO_FAILURE)) {
3986                 SF_DEBUG(2, (sf, CE_CONT,
3987                     "!INQUIRY to al_pa %x lun %" PRIx64
3988                     " succeeded\n",
3989                     privp->dest_nport_id, SCSA_LUN(target)));
3991                 (void) ddi_dma_sync(privp->data_dma_handle,
3992                     (off_t)0, (size_t)0,
3993                     DDI_DMA_SYNC_FORKERNEL);
3995                 mutex_enter(&sf->sf_mutex);
3997                 if (sf->sf_lip_cnt == privp->lip_cnt) {
3998                     mutex_enter(&target->sft_mutex);
3999                     target->sft_device_type =
4000                         prt->inq_dtype;
4001                     bcopy(prt, &target->sft_inq,
4002                         sizeof (*prt));
4003                     mutex_exit(&target->sft_mutex);
4004                     sf->sf_device_count--;
4005                     ASSERT(sf->sf_device_count >= 0);
4006                     if (sf->sf_device_count == 0) {
4007                         sf_finish_init(sf,
4008                             privp->lip_cnt);
4009                     }
4010                 }
4011                 mutex_exit(&sf->sf_mutex);
4012                 sf_els_free(fpkt);
4013                 return;
4014             }
4015         } else if ((rsp->fcp_u.fcp_status.scsi_status ==
4016             STATUS_BUSY) ||
4017             (rsp->fcp_u.fcp_status.scsi_status == STATUS_QFULL) ||
4018             (rsp->fcp_u.fcp_status.scsi_status == STATUS_CHECK)) {

```

```

4019         delayed_retry = TRUE;
4020     }
4021     } else {
4022         SF_DEBUG(2, (sf, CE_CONT, "!INQUIRY to al_pa %x fc status %x",
4023             privp->dest_nport_id, fpkt->fcal_pkt_status));
4024     }
4026     if (++(privp->retries) < sf_els_retries ||
4027         (delayed_retry && privp->retries < SF_BSY_RETRIES)) {
4028         if (fpkt->fcal_pkt_status == FCAL_STATUS_MAX_XCHG_EXCEEDED) {
4029             tsf = sf->sf_sibling;
4030             if (tsf != NULL) {
4031                 mutex_enter(&tsf->sf_cmd_mutex);
4032                 tsf->sf_flag = 1;
4033                 tsf->sf_throttle = SF_DECR_DELTA;
4034                 mutex_exit(&tsf->sf_cmd_mutex);
4035             }
4036             delayed_retry = 1;
4037         }
4038         if (delayed_retry) {
4039             privp->retries--;
4040             privp->timeout = sf_watchdog_time + SF_BSY_TIMEOUT;
4041             privp->delayed_retry = TRUE;
4042         } else {
4043             privp->timeout = sf_watchdog_time + SF_FCP_TIMEOUT;
4044         }
4046         privp->prev = NULL;
4047         mutex_enter(&sf->sf_mutex);
4048         if (privp->lip_cnt == sf->sf_lip_cnt) {
4049             if (!delayed_retry) {
4050                 SF_DEBUG(1, (sf, CE_WARN,
4051                     "INQUIRY to al_pa %x failed, retrying",
4052                     privp->dest_nport_id));
4053             }
4054             privp->next = sf->sf_els_list;
4055             if (sf->sf_els_list != NULL) {
4056                 sf->sf_els_list->prev = privp;
4057             }
4058             sf->sf_els_list = privp;
4059             mutex_exit(&sf->sf_mutex);
4060             /* if not delayed call transport to send a pkt */
4061             if (!delayed_retry &&
4062                 (soc_transport(sf->sf_sochandle, fpkt,
4063                     FCAL_NOSLEEP, CQ_REQUEST_1) !=
4064                     FCAL_TRANSPORT_SUCCESS)) {
4065                 mutex_enter(&sf->sf_mutex);
4066                 if (privp->prev != NULL) {
4067                     privp->prev->next = privp->next;
4068                 }
4069                 if (privp->next != NULL) {
4070                     privp->next->prev = privp->prev;
4071                 }
4072                 if (sf->sf_els_list == privp) {
4073                     sf->sf_els_list = privp->next;
4074                 }
4075                 mutex_exit(&sf->sf_mutex);
4076                 goto fail;
4077             }
4078             return;
4079         }
4080         mutex_exit(&sf->sf_mutex);
4081     } else {
4082 fail:
4083         mutex_enter(&sf->sf_mutex);
4084         if (sf->sf_lip_cnt == privp->lip_cnt) {

```

```

4085     sf_offline_target(sf, target);
4086     sf_log(sf, CE_NOTE,
4087           "INQUIRY to target 0x%x lun %" PRIx64 " failed. "
4088           "Retry Count: %d\n",
4089           sf_alpa_to_switch[privp->dest_nport_id],
4090           SCSI_LUN(target),
4091           privp->retries);
4092     sf->sf_device_count--;
4093     ASSERT(sf->sf_device_count >= 0);
4094     if (sf->sf_device_count == 0) {
4095         sf_finish_init(sf, privp->lip_cnt);
4096     }
4097     }
4098     mutex_exit(&sf->sf_mutex);
4099 }
4100 sf_els_free(fpkt);
4101 }

4104 static void
4105 sf_finish_init(struct sf *sf, int lip_cnt)
4106 {
4107     int             i;                /* loop index */
4108     int             cflag;
4109     struct sf_target *target;        /* current target */
4110     dev_info_t      *dip;
4111     struct sf_hp_elem *elem;        /* hotplug element created */

4113     SF_DEBUG(1, (sf, CE_WARN, "!sf_finish_init\n"));
4114     ASSERT(mutex_owned(&sf->sf_mutex));

4116     /* scan all hash queues */
4117     for (i = 0; i < SF_NUM_HASH_QUEUES; i++) {
4118         target = sf->sf_wnn_lists[i];
4119         while (target != NULL) {
4120             mutex_enter(&target->sft_mutex);

4122             /* see if target is not offline */
4123             if ((target->sft_state & SF_TARGET_OFFLINE)) {
4124                 /*
4125                  * target already offline
4126                  */
4127                 mutex_exit(&target->sft_mutex);
4128                 goto next_entry;
4129             }

4131             /*
4132              * target is not already offline -- see if it has
4133              * already been marked as ready to go offline
4134              */
4135             if (target->sft_state & SF_TARGET_MARK) {
4136                 /*
4137                  * target already marked, so take it offline
4138                  */
4139                 mutex_exit(&target->sft_mutex);
4140                 sf_offline_target(sf, target);
4141                 goto next_entry;
4142             }

4144             /* clear target busy flag */
4145             target->sft_state &= ~SF_TARGET_BUSY;

4147             /* is target init not yet done ?? */
4148             cflag = !(target->sft_state & SF_TARGET_INIT_DONE);

4150             /* get pointer to target dip */

```

```

4151         dip = target->sft_dip;

4153         mutex_exit(&target->sft_mutex);
4154         mutex_exit(&sf->sf_mutex);

4156         if (cflag && (dip == NULL)) {
4157             /*
4158              * target init not yet done &&
4159              * devinfo not yet created
4160              */
4161             sf_create_devinfo(sf, target, lip_cnt);
4162             mutex_enter(&sf->sf_mutex);
4163             goto next_entry;
4164         }

4166         /*
4167          * target init already done || devinfo already created
4168          */
4169         ASSERT(dip != NULL);
4170         if (!sf_create_props(dip, target, lip_cnt)) {
4171             /* a problem creating properties */
4172             mutex_enter(&sf->sf_mutex);
4173             goto next_entry;
4174         }

4176         /* create a new element for the hotplug list */
4177         if ((elem = kmem_zalloc(sizeof (struct sf_hp_elem),
4178                                KM_NOSLEEP)) != NULL) {

4180             /* fill in the new element */
4181             elem->dip = dip;
4182             elem->target = target;
4183             elem->what = SF_ONLINE;

4185             /* add the new element into the hotplug list */
4186             mutex_enter(&sf->sf_hp_daemon_mutex);
4187             if (sf->sf_hp_elem_tail != NULL) {
4188                 sf->sf_hp_elem_tail->next = elem;
4189                 sf->sf_hp_elem_tail = elem;
4190             } else {
4191                 /* this is the first element in list */
4192                 sf->sf_hp_elem_head =
4193                     sf->sf_hp_elem_tail =
4194                         elem;
4195             }
4196             cv_signal(&sf->sf_hp_daemon_cv);
4197             mutex_exit(&sf->sf_hp_daemon_mutex);
4198         } else {
4199             /* could not allocate memory for element ?? */
4200             (void) ndi_devi_online_async(dip, 0);
4201         }

4203         mutex_enter(&sf->sf_mutex);

4205     next_entry:
4206         /* ensure no new LIPs have occurred */
4207         if (sf->sf_lip_cnt != lip_cnt) {
4208             return;
4209         }
4210         target = target->sft_next;
4211     }

4213     /* done scanning all targets in this queue */
4214 }

4216     /* done with all hash queues */

```

```

4218     sf->sf_state = SF_STATE_ONLINE;
4219     sf->sf_online_timer = 0;
4220 }

4223 /*
4224  * create devinfo node
4225  */
4226 static void
4227 sf_create_devinfo(struct sf *sf, struct sf_target *target, int lip_cnt)
4228 {
4229     dev_info_t      *cdip = NULL;
4230     char            *nname = NULL;
4231     char            **compatible = NULL;
4232     int             ncompatible;
4233     struct scsi_inquiry *inq = &target->sft_inq;
4234     char            *scsi_binding_set;

4236     /* get the 'scsi-binding-set' property */
4237     if (ddi_prop_lookup_string(DDI_DEV_T_ANY, sf->sf_dip,
4238         DDI_PROP_NOTPROM | DDI_PROP_DONTPASS, "scsi-binding-set",
4239         &scsi_binding_set) != DDI_PROP_SUCCESS)
4240         scsi_binding_set = NULL;

4242     /* determine the node name and compatible */
4243     scsi_hba_nodename_compatible_get(inq, scsi_binding_set,
4244         inq->inq_dtype, NULL, &nname, &compatible, &ncompatible);
4245     if (scsi_binding_set)
4246         ddi_prop_free(scsi_binding_set);

4248     /* if nodename can't be determined then print a message and skip it */
4249     if (nname == NULL) {
4250 #ifndef RAID_LUNS
4251         sf_log(sf, CE_WARN, "%s%d: no driver for device "
4252             "%w%02x%02x%02x%02x%02x%02x%02x,%x\n"
4253             "    compatible: %s",
4254             ddi_driver_name(sf->sf_dip), ddi_get_instance(sf->sf_dip),
4255             target->sft_port_wnn[0], target->sft_port_wnn[1],
4256             target->sft_port_wnn[2], target->sft_port_wnn[3],
4257             target->sft_port_wnn[4], target->sft_port_wnn[5],
4258             target->sft_port_wnn[6], target->sft_port_wnn[7],
4259             target->sft_lun.l, *compatible);
4260 #else
4261         sf_log(sf, CE_WARN, "%s%d: no driver for device "
4262             "%w%02x%02x%02x%02x%02x%02x%02x,%x\n"
4263             "    compatible: %s",
4264             ddi_driver_name(sf->sf_dip), ddi_get_instance(sf->sf_dip),
4265             target->sft_port_wnn[0], target->sft_port_wnn[1],
4266             target->sft_port_wnn[2], target->sft_port_wnn[3],
4267             target->sft_port_wnn[4], target->sft_port_wnn[5],
4268             target->sft_port_wnn[6], target->sft_port_wnn[7],
4269             target->sft_raid_lun, *compatible);
4270 #endif
4271         goto fail;
4272     }

4274     /* allocate the node */
4275     if (ndi_devi_alloc(sf->sf_dip, nname,
4276         DEVI_SID_NODEID, &cdip) != NDI_SUCCESS) {
4277         goto fail;
4278     }

4280     /* decorate the node with compatible */
4281     if (ndi_prop_update_string_array(DDI_DEV_T_NONE, cdip,
4282         "compatible", compatible, ncompatible) != DDI_PROP_SUCCESS) {

```

```

4283         goto fail;
4284     }

4286     /* add addressing properties to the node */
4287     if (sf_create_props(cdip, target, lip_cnt) != 1) {
4288         goto fail;
4289     }

4291     mutex_enter(&target->sft_mutex);
4292     if (target->sft_dip != NULL) {
4293         mutex_exit(&target->sft_mutex);
4294         goto fail;
4295     }
4296     target->sft_dip = cdip;
4297     mutex_exit(&target->sft_mutex);

4299     if (ndi_devi_online_async(cdip, 0) != DDI_SUCCESS) {
4300         goto fail;
4301     }

4303     scsi_hba_nodename_compatible_free(nname, compatible);
4304     return;

4306 fail:
4307     scsi_hba_nodename_compatible_free(nname, compatible);
4308     if (cdip != NULL) {
4309         (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip, NODE_WWN_PROP);
4310         (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip, PORT_WWN_PROP);
4311         (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip, LIP_CNT_PROP);
4312         (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip, TARGET_PROP);
4313         (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip, LUN_PROP);
4314         if (ndi_devi_free(cdip) != NDI_SUCCESS) {
4315             sf_log(sf, CE_WARN, "ndi_devi_free failed\n");
4316         } else {
4317             mutex_enter(&target->sft_mutex);
4318             if (cdip == target->sft_dip) {
4319                 target->sft_dip = NULL;
4320             }
4321             mutex_exit(&target->sft_mutex);
4322         }
4323     }
4324 }

4326 /*
4327  * create required properties, returning TRUE iff we succeed, else
4328  * returning FALSE
4329  */
4330 static int
4331 sf_create_props(dev_info_t *cdip, struct sf_target *target, int lip_cnt)
4332 {
4333     int tgt_id = sf_alpa_to_switch[target->sft_al_pa];

4336     if (ndi_prop_update_byte_array(DDI_DEV_T_NONE,
4337         cdip, NODE_WWN_PROP, target->sft_node_wnn, FC_WWN_SIZE) !=
4338         DDI_PROP_SUCCESS) {
4339         return (FALSE);
4340     }

4342     if (ndi_prop_update_byte_array(DDI_DEV_T_NONE,
4343         cdip, PORT_WWN_PROP, target->sft_port_wnn, FC_WWN_SIZE) !=
4344         DDI_PROP_SUCCESS) {
4345         return (FALSE);
4346     }

4348     if (ndi_prop_update_int(DDI_DEV_T_NONE,

```

```

4349         cdip, LIP_CNT_PROP, lip_cnt) != DDI_PROP_SUCCESS) {
4350             return (FALSE);
4351         }
4353         if (ndi_prop_update_int(DDI_DEV_T_NONE,
4354             cdip, TARGET_PROP, tgt_id) != DDI_PROP_SUCCESS) {
4355             return (FALSE);
4356         }
4358 #ifndef RAID_LUNS
4359         if (ndi_prop_update_int(DDI_DEV_T_NONE,
4360             cdip, LUN_PROP, target->sft_lun.l) != DDI_PROP_SUCCESS) {
4361             return (0);
4362         }
4363 #else
4364         if (ndi_prop_update_int(DDI_DEV_T_NONE,
4365             cdip, LUN_PROP, target->sft_raid_lun) != DDI_PROP_SUCCESS) {
4366             return (0);
4367         }
4368 #endif
4370         return (TRUE);
4371     }
4374 /*
4375  * called by the transport to offline a target
4376  */
4377 /* ARGSUSED */
4378 static void
4379 sf_offline_target(struct sf *sf, struct sf_target *target)
4380 {
4381     dev_info_t *dip;
4382     struct sf_target *next_target = NULL;
4383     struct sf_hp_elem *elem;
4385     ASSERT(mutex_owned(&sf->sf_mutex));
4387     if (sf_core && (sf_core & SF_CORE_OFFLINE_TARGET)) {
4388         (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
4389         sf_core = 0;
4390     }
4392     while (target != NULL) {
4393         sf_log(sf, CE_NOTE,
4394             "!target 0x%x al_pa 0x%x lun %" PRIx64 " offlined\n",
4395             sf_alpa_to_switch[target->sft_al_pa],
4396             target->sft_al_pa, SCSA_LUN(target));
4397         mutex_enter(&target->sft_mutex);
4398         target->sft_state &= ~(SF_TARGET_BUSY|SF_TARGET_MARK);
4399         target->sft_state |= SF_TARGET_OFFLINE;
4400         mutex_exit(&target->sft_mutex);
4401         mutex_exit(&sf->sf_mutex);
4403         /* XXXX if this is LUN 0, offline all other LUNs */
4404         if (next_target || target->sft_lun.l == 0)
4405             next_target = target->sft_next_lun;
4407         /* abort all cmds for this target */
4408         sf_abort_all(sf, target, FALSE, sf->sf_lip_cnt, FALSE);
4410         mutex_enter(&sf->sf_mutex);
4411         mutex_enter(&target->sft_mutex);
4412         if (target->sft_state & SF_TARGET_INIT_DONE) {
4413             dip = target->sft_dip;
4414             mutex_exit(&target->sft_mutex);

```

```

4415         mutex_exit(&sf->sf_mutex);
4416         (void) ndi_prop_remove(DDI_DEV_T_NONE, dip,
4417             TARGET_PROP);
4418         (void) ndi_event_retrieve_cookie(sf->sf_event_hdl,
4419             dip, FCAL_REMOVE_EVENT, &sf_remove_eid,
4420             NDI_EVENT_NOPASS);
4421         (void) ndi_event_run_callbacks(sf->sf_event_hdl,
4422             target->sft_dip, sf_remove_eid, NULL);
4424         elem = kmem_zalloc(sizeof (struct sf_hp_elem),
4425             KM_NOSLEEP);
4426         if (elem != NULL) {
4427             elem->dip = dip;
4428             elem->target = target;
4429             elem->what = SF_OFFLINE;
4430             mutex_enter(&sf->sf_hp_daemon_mutex);
4431             if (sf->sf_hp_elem_tail != NULL) {
4432                 sf->sf_hp_elem_tail->next = elem;
4433                 sf->sf_hp_elem_tail = elem;
4434             } else {
4435                 sf->sf_hp_elem_head =
4436                     sf->sf_hp_elem_tail =
4437                         elem;
4438             }
4439             cv_signal(&sf->sf_hp_daemon_cv);
4440             mutex_exit(&sf->sf_hp_daemon_mutex);
4441         } else {
4442             /* don't do NDI_DEVI_REMOVE for now */
4443             if (ndi_devi_offline(dip, 0) != NDI_SUCCESS) {
4444                 SF_DEBUG(1, (sf, CE_WARN,
4445                     "target %x lun %" PRIx64 " ", "
4446                     "device offline failed",
4447                     sf_alpa_to_switch[target->
4448                         sft_al_pa],
4449                     SCSA_LUN(target)));
4450             } else {
4451                 SF_DEBUG(1, (sf, CE_NOTE,
4452                     "target %x, lun %" PRIx64 " ", "
4453                     "device offline succeeded\n",
4454                     sf_alpa_to_switch[target->
4455                         sft_al_pa],
4456                     SCSA_LUN(target)));
4457             }
4458         }
4459         mutex_enter(&sf->sf_mutex);
4460     } else {
4461         mutex_exit(&target->sft_mutex);
4462     }
4463     target = next_target;
4464 }
4465 }
4468 /*
4469  * routine to get/set a capability
4470  */
4471 * returning:
4472 *     1 (TRUE)         boolean capability is true (on get)
4473 *     0 (FALSE)       invalid capability, can't set capability (on set),
4474 *                    or boolean capability is false (on get)
4475 *    -1 (UNDEFINED)   can't find capability (SCSA) or unsupported capability
4476 *     3                when getting SCSI version number
4477 *     AL_PA           when getting port initiator ID
4478 */
4479 static int
4480 sf_commoncap(struct scsi_address *ap, char *cap,

```

```

4481     int val, int tgtonly, int doset)
4482 {
4483     struct sf *sf = ADDR2SF(ap);
4484     int cidx;
4485     int rval = FALSE;

4488     if (cap == NULL) {
4489         SF_DEBUG(3, (sf, CE_WARN, "sf_commoncap: invalid arg"));
4490         return (rval);
4491     }

4493     /* get index of capability string */
4494     if ((cidx = scsi_hba_lookup_capstr(cap)) == -1) {
4495         /* can't find capability */
4496         return (UNDEFINED);
4497     }

4499     if (doset) {
4500         /*
4501          * Process setcap request.
4502          */

4504         /*
4505          * At present, we can only set binary (0/1) values
4506          */
4507         switch (cidx) {
4508             case SCSI_CAP_ARQ:      /* can't set this capability */
4509                 break;
4510             default:
4511                 SF_DEBUG(3, (sf, CE_WARN,
4512                     "sf_setcap: unsupported %d", cidx));
4513                 rval = UNDEFINED;
4514                 break;
4515         }

4517         SF_DEBUG(4, (sf, CE_NOTE,
4518             "set cap: cap=%s,val=0x%x,tgtonly=0x%x"
4519             ",doset=0x%x,rval=%d\n",
4520             cap, val, tgtonly, doset, rval));

4522     } else {
4523         /*
4524          * Process getcap request.
4525          */
4526         switch (cidx) {
4527             case SCSI_CAP_DMA_MAX:
4528                 break; /* don't have this capability */
4529             case SCSI_CAP_INITIATOR_ID:
4530                 rval = sf->sf_al_pa;
4531                 break;
4532             case SCSI_CAP_ARQ:
4533                 rval = TRUE; /* do have this capability */
4534                 break;
4535             case SCSI_CAP_RESET_NOTIFICATION:
4536             case SCSI_CAP_TAGGED_QING:
4537                 rval = TRUE; /* do have this capability */
4538                 break;
4539             case SCSI_CAP_SCSI_VERSION:
4540                 rval = 3;
4541                 break;
4542             case SCSI_CAP_INTERCONNECT_TYPE:
4543                 rval = INTERCONNECT_FIBRE;
4544                 break;
4545             default:
4546                 SF_DEBUG(4, (sf, CE_WARN,

```

```

4547         "sf_scsi_getcap: unsupported"));
4548         rval = UNDEFINED;
4549         break;
4550     }
4551     SF_DEBUG(4, (sf, CE_NOTE,
4552         "get cap: cap=%s,val=0x%x,tgtonly=0x%x,"
4553         "doset=0x%x,rval=%d\n",
4554         cap, val, tgtonly, doset, rval));
4555     }

4557     return (rval);
4558 }

4561 /*
4562  * called by the transport to get a capability
4563  */
4564 static int
4565 sf_getcap(struct scsi_address *ap, char *cap, int whom)
4566 {
4567     return (sf_commoncap(ap, cap, 0, whom, FALSE));
4568 }

4571 /*
4572  * called by the transport to set a capability
4573  */
4574 static int
4575 sf_setcap(struct scsi_address *ap, char *cap, int value, int whom)
4576 {
4577     return (sf_commoncap(ap, cap, value, whom, TRUE));
4578 }

4581 /*
4582  * called by the transport to abort a target
4583  */
4584 static int
4585 sf_abort(struct scsi_address *ap, struct scsi_pkt *pkt)
4586 {
4587     struct sf *sf = ADDR2SF(ap);
4588     struct sf_target *target = ADDR2TARGET(ap);
4589     struct sf_pkt *cmd, *ncmd, *pcmd;
4590     struct fc_al_packet *fpkt;
4591     int rval = 0, t, my_rval = FALSE;
4592     int old_target_state;
4593     int lip_cnt;
4594     int tgt_id;
4595     fc_frame_header_t *hp;
4596     int deferred_destroy;

4598     deferred_destroy = 0;

4600     if (pkt != NULL) {
4601         cmd = PKT2CMD(pkt);
4602         fpkt = cmd->cmd_fp_pkt;
4603         SF_DEBUG(2, (sf, CE_NOTE, "sf_abort packet %p\n",
4604             (void *)fpkt));
4605         pcmd = NULL;
4606         mutex_enter(&sf->sf_cmd_mutex);
4607         ncmd = sf->sf_pkt_head;
4608         while (ncmd != NULL) {
4609             if (ncmd == cmd) {
4610                 if (pcmd != NULL) {
4611                     pcmd->cmd_next = cmd->cmd_next;
4612                 } else {

```

```

4613         sf->sf_pkt_head = cmd->cmd_next;
4614     }
4615     cmd->cmd_flags &= ~CFLAG_IN_QUEUE;
4616     cmd->cmd_state = SF_STATE_IDLE;
4617     pkt->pkt_reason = CMD_ABORTED;
4618     pkt->pkt_statistics |= STAT_ABORTED;
4619     my_rval = TRUE;
4620     break;
4621 } else {
4622     pcmd = ncmd;
4623     ncmd = ncmd->cmd_next;
4624 }
4625 }
4626 mutex_exit(&sf->sf_cmd_mutex);
4627 if (ncmd == NULL) {
4628     mutex_enter(&cmd->cmd_abort_mutex);
4629     if (cmd->cmd_state == SF_STATE_ISSUED) {
4630         cmd->cmd_state = SF_STATE_ABORTING;
4631         cmd->cmd_timeout = sf_watchdog_time + 20;
4632         mutex_exit(&cmd->cmd_abort_mutex);
4633         /* call transport to abort command */
4634         if (((rval = soc_abort(sf->sf_sochandle,
4635             sf->sf_socp, sf->sf_sochandle->fcal_portno,
4636             fpkt, 1)) == FCAL_ABORTED) ||
4637             (rval == FCAL_ABORT_FAILED)) {
4638             my_rval = TRUE;
4639             pkt->pkt_reason = CMD_ABORTED;
4640             pkt->pkt_statistics |= STAT_ABORTED;
4641             cmd->cmd_state = SF_STATE_IDLE;
4642         } else if (rval == FCAL_BAD_ABORT) {
4643             cmd->cmd_timeout = sf_watchdog_time
4644                 + 20;
4645             my_rval = FALSE;
4646         } else {
4647             SF_DEBUG(1, (sf, CE_NOTE,
4648                 "Command Abort failed\n"));
4649         }
4650     } else {
4651         mutex_exit(&cmd->cmd_abort_mutex);
4652     }
4653 }
4654 } else {
4655     SF_DEBUG(2, (sf, CE_NOTE, "sf_abort target\n"));
4656     mutex_enter(&sf->sf_mutex);
4657     lip_cnt = sf->sf_lip_cnt;
4658     mutex_enter(&target->sft_mutex);
4659     if (target->sft_state & (SF_TARGET_BUSY |
4660         SF_TARGET_OFFLINE)) {
4661         mutex_exit(&target->sft_mutex);
4662         return (rval);
4663     }
4664     old_target_state = target->sft_state;
4665     target->sft_state |= SF_TARGET_BUSY;
4666     mutex_exit(&target->sft_mutex);
4667     mutex_exit(&sf->sf_mutex);
4669
4670     if ((pkt = sf_scsi_init_pkt(ap, NULL, NULL, 0,
4671         0, 0, 0, NULL, 0)) != NULL) {
4672         cmd = PKT2CMD(pkt);
4673         cmd->cmd_block->fcp_cntl.cntl_abort_tsk = 1;
4674         cmd->cmd_fp_pkt->fcal_pkt_comp = NULL;
4675         cmd->cmd_pkt->pkt_flags |= FLAG_NOINTR;
4677
4678         /* prepare the packet for transport */
4679         if (sf_prepare_pkt(sf, cmd, target) == TRAN_ACCEPT) {

```

```

4680         cmd->cmd_state = SF_STATE_ISSUED;
4681         /*
4682          * call transport to send a pkt polled
4683          *
4684          * if that fails call the transport to abort it
4685          */
4686         if (soc_transport_poll(sf->sf_sochandle,
4687             cmd->cmd_fp_pkt, SF_ABORT_TIMEOUT,
4688             CQ_REQUEST_1) == FCAL_TRANSPORT_SUCCESS) {
4689             (void) ddi_dma_sync(
4690                 cmd->cmd_cr_pool->rsp_dma_handle,
4691                 (off_t)
4692                 ((caddr_t)cmd->cmd_rsp_block -
4693                 cmd->cmd_cr_pool->rsp_base),
4694                 FCP_MAX_RSP_IU_SIZE,
4695                 DDI_DMA_SYNC_FORKERNEL);
4696             if (((struct fcp_rsp_info *)
4697                 (&cmd->cmd_rsp_block->
4698                 fcp_response_len + 1))->
4699                 rsp_code == FCP_NO_FAILURE) {
4700                 /* abort cmds for this targ */
4701                 sf_abort_all(sf, target, TRUE,
4702                     lip_cnt, TRUE);
4703             } else {
4704                 hp = &cmd->cmd_fp_pkt->
4705                     fcal_socal_request.
4706                     sr_fc_frame_hdr;
4707                 tgt_id = sf_alpa_to_switch[
4708                     (uchar_t)hp->d_id];
4709                 sf->sf_stats.tstats[tgt_id].
4710                     task_mgmt_failures++;
4711                 SF_DEBUG(1, (sf, CE_NOTE,
4712                     "Target %d Abort Task "
4713                     "Set failed\n", hp->d_id));
4714             }
4715         } else {
4716             mutex_enter(&cmd->cmd_abort_mutex);
4717             if (cmd->cmd_state == SF_STATE_ISSUED) {
4718                 cmd->cmd_state = SF_STATE_ABORTING;
4719                 cmd->cmd_timeout = sf_watchdog_time
4720                     + 20;
4721                 mutex_exit(&cmd->cmd_abort_mutex);
4722                 if ((t = soc_abort(sf->sf_sochandle,
4723                     sf->sf_socp, sf->sf_sochandle->
4724                     fcal_portno, cmd->cmd_fp_pkt, 1)) !=
4725                     FCAL_ABORTED &&
4726                     (t != FCAL_ABORT_FAILED)) {
4727                     sf_log(sf, CE_NOTE,
4728                         "sf_abort failed, "
4729                         "initiating LIP\n");
4730                     sf_force_lip(sf);
4731                     deferred_destroy = 1;
4732                 }
4733             } else {
4734                 mutex_exit(&cmd->cmd_abort_mutex);
4735             }
4736         }
4737     }
4738     if (!deferred_destroy) {
4739         cmd->cmd_fp_pkt->fcal_pkt_comp =
4740             sf_cmd_callback;
4741         cmd->cmd_block->fcp_cntl.cntl_abort_tsk = 0;
4742         sf_scsi_destroy_pkt(ap, pkt);
4743         my_rval = TRUE;
4744     }

```



```

4877 mutex_enter(&cmd->cmd_abort_mutex);
4878 if (cmd->cmd_state == SF_STATE_ISSUED) {
4879 /* call the transport to abort a cmd */
4880 cmd->cmd_timeout = sf_watchdog_time
4881 + 20;
4882 cmd->cmd_state = SF_STATE_ABORTING;
4883 mutex_exit(&cmd->cmd_abort_mutex);
4884 if (((t = soc_abort(sf->sf_sochandle,
4885 sf->sf_socp,
4886 sf->sf_sochandle->fcml_portno,
4887 cmd->cmd_fp_pkt, 1)) !=
4888 FCAL_ABORTED) &&
4889 (t != FCAL_ABORT_FAILED)) {
4890 sf_log(sf, CE_NOTE,
4891 " !sf%d: Target 0x%x Reset "
4892 "failed. Abort Failed, "
4893 "forcing LIP\n",
4894 ddi_get_instance(
4895 sf->sf_dip),
4896 sf_alpa_to_switch[
4897 target->sft_al_pa]);
4898 sf_force_lip(sf);
4899 rval = TRUE;
4900 deferred_destroy = 1;
4901 } else {
4902 mutex_exit
4903 (&cmd->cmd_abort_mutex);
4904 }
4905 }
4906 }
4907 /*
4908 * Defer releasing the packet if we abort returned with
4909 * a BAD_ABORT or timed out, because there is a
4910 * possibility that the ucode might return it.
4911 * We wait for at least 20s and let it be released
4912 * by the sf_watch thread
4913 */
4914 if (!deferred_destroy) {
4915 cmd->cmd_block->fcp_cntl.cntl_reset = 0;
4916 cmd->cmd_fp_pkt->fcml_pkt_comp =
4917 sf_cmd_callback;
4918 cmd->cmd_state = SF_STATE_IDLE;
4919 /* for cache */
4920 sf_scsi_destroy_pkt(ap, pkt);
4921 }
4922 } else {
4923 cmn_err(CE_WARN, " !sf%d: Target 0x%x Reset Failed. "
4924 "Resource allocation error.\n",
4925 ddi_get_instance(sf->sf_dip),
4926 sf_alpa_to_switch[target->sft_al_pa]);
4927 }
4928 mutex_enter(&sf->sf_mutex);
4929 if ((rval == TRUE) && (lip_cnt == sf->sf_lip_cnt)) {
4930 p->target = target;
4931 p->lip_cnt = lip_cnt;
4932 p->timeout = ddi_get_lbolt() +
4933 drv_usec2hz(SF_TARGET_RESET_DELAY);
4934 p->next = sf->sf_reset_list;
4935 sf->sf_reset_list = p;
4936 mutex_exit(&sf->sf_mutex);
4937 mutex_enter(&sf_global_mutex);
4938 if (sf_reset_timeout_id == 0) {
4939 sf_reset_timeout_id = timeout(
4940 sf_check_reset_delay, NULL,
4941 drv_usec2hz(SF_TARGET_RESET_DELAY));
4942

```

```

4943 }
4944 mutex_exit(&sf_global_mutex);
4945 } else {
4946 if (lip_cnt == sf->sf_lip_cnt) {
4947 mutex_enter(&target->sft_mutex);
4948 target->sft_state &= ~SF_TARGET_BUSY;
4949 for (ntarget = target->sft_next_lun;
4950 ntarget;
4951 ntarget = ntarget->sft_next_lun) {
4952 mutex_enter(&ntarget->sft_mutex);
4953 ntarget->sft_state &= ~SF_TARGET_BUSY;
4954 mutex_exit(&ntarget->sft_mutex);
4955 }
4956 mutex_exit(&target->sft_mutex);
4957 }
4958 mutex_exit(&sf->sf_mutex);
4959 kmem_free(p, sizeof(struct sf_reset_list));
4960 } else {
4961 mutex_enter(&sf->sf_mutex);
4962 if ((sf->sf_state == SF_STATE_OFFLINE) &&
4963 (sf_watchdog_time < sf->sf_timer)) {
4964 /*
4965 * We are currently in a lip, so let this one
4966 * finish before forcing another one.
4967 */
4968 mutex_exit(&sf->sf_mutex);
4969 return (TRUE);
4970 }
4971 mutex_exit(&sf->sf_mutex);
4972 sf_log(sf, CE_NOTE, " !sf:Target driver initiated lip\n");
4973 sf_force_lip(sf);
4974 rval = TRUE;
4975 }
4976 return (rval);
4977 }
4978 }

4981 /*
4982 * abort all commands for a target
4983 *
4984 * if try_abort is set then send an abort
4985 * if abort is set then this is abort, else this is a reset
4986 */
4987 static void
4988 sf_abort_all(struct sf *sf, struct sf_target *target, int abort, int
4989 lip_cnt, int try_abort)
4990 {
4991 struct sf_target *ntarget;
4992 struct sf_pkt *cmd, *head = NULL, *tail = NULL, *pcmd = NULL, *tcmd;
4993 struct fcml_packet *fpkt;
4994 struct scsi_pkt *pkt;
4995 int rval = FCAL_ABORTED;

4997 /*
4998 * First pull all commands for all LUNs on this target out of the
4999 * overflow list. We can tell it's the same target by comparing
5000 * the node WWN.
5001 */
5002 mutex_enter(&sf->sf_mutex);
5003 if (lip_cnt == sf->sf_lip_cnt) {
5004 mutex_enter(&sf->sf_cmd_mutex);
5005 cmd = sf->sf_pkt_head;
5006 while (cmd != NULL) {
5007 ntarget = ADDR2TARGET(&cmd->cmd_pkt->
5008 pkt_address);

```

```

5009         if (ntarget == target) {
5010             if (pcmd != NULL)
5011                 pcmd->cmd_next = cmd->cmd_next;
5012             else
5013                 sf->sf_pkt_head = cmd->cmd_next;
5014             if (sf->sf_pkt_tail == cmd) {
5015                 sf->sf_pkt_tail = pcmd;
5016                 if (pcmd != NULL)
5017                     pcmd->cmd_next = NULL;
5018             }
5019             tcmd = cmd->cmd_next;
5020             if (head == NULL) {
5021                 head = cmd;
5022                 tail = cmd;
5023             } else {
5024                 tail->cmd_next = cmd;
5025                 tail = cmd;
5026             }
5027             cmd->cmd_next = NULL;
5028             cmd = tcmd;
5029         } else {
5030             pcmd = cmd;
5031             cmd = cmd->cmd_next;
5032         }
5033     }
5034     mutex_exit(&sf->sf_cmd_mutex);
5035 }
5036 mutex_exit(&sf->sf_mutex);

5038 /*
5039  * Now complete all the commands on our list.  In the process,
5040  * the completion routine may take the commands off the target
5041  * lists.
5042  */
5043 cmd = head;
5044 while (cmd != NULL) {
5045     pkt = cmd->cmd_pkt;
5046     if (abort) {
5047         pkt->pkt_reason = CMD_ABORTED;
5048         pkt->pkt_statistics |= STAT_ABORTED;
5049     } else {
5050         pkt->pkt_reason = CMD_RESET;
5051         pkt->pkt_statistics |= STAT_DEV_RESET;
5052     }
5053     cmd->cmd_flags &= ~CFLAG_IN_QUEUE;
5054     cmd->cmd_state = SF_STATE_IDLE;
5055     cmd = cmd->cmd_next;
5056     /*
5057      * call the packet completion routine only for
5058      * non-pollled commands. Ignore the polled commands as
5059      * they timeout and will be handled differently
5060      */
5061     if ((pkt->pkt_comp) && !(pkt->pkt_flags & FLAG_NOINTR))
5062         (*pkt->pkt_comp)(pkt);
5064 }

5066 /*
5067  * Finally get all outstanding commands for each LUN, and abort them if
5068  * they've been issued, and call the completion routine.
5069  * For the case where sf_offline_target is called from sf_watch
5070  * due to a Offline Timeout, it is quite possible that the soc
5071  * ucode is hosed and therefore cannot return the commands.
5072  * Clear up all the issued commands as well.
5073  * Try_abort will be false only if sf_abort_all is coming from
5074  * sf_target_offline.

```

```

5075     */
5077     if (try_abort || sf->sf_state == SF_STATE_OFFLINE) {
5078         mutex_enter(&target->sft_pkt_mutex);
5079         cmd = tcmd = target->sft_pkt_head;
5080         while (cmd != (struct sf_pkt *)&target->sft_pkt_head) {
5081             fpkt = cmd->cmd_fp_pkt;
5082             pkt = cmd->cmd_pkt;
5083             mutex_enter(&cmd->cmd_abort_mutex);
5084             if ((cmd->cmd_state == SF_STATE_ISSUED) &&
5085                 (fpkt->fcml_cmd_state &
5086                  FCML_CMD_IN_TRANSPORT) &&
5087                 ((fpkt->fcml_cmd_state & FCML_CMD_COMPLETE) ==
5088                  0) && !(pkt->pkt_flags & FLAG_NOINTR)) {
5089                 cmd->cmd_state = SF_STATE_ABORTING;
5090                 cmd->cmd_timeout = sf_watchdog_time +
5091                     cmd->cmd_pkt->pkt_time + 20;
5092                 mutex_exit(&cmd->cmd_abort_mutex);
5093                 mutex_exit(&target->sft_pkt_mutex);
5094                 if (try_abort) {
5095                     /* call the transport to abort a pkt */
5096                     rval = soc_abort(sf->sf_sochandle,
5097                                     sf->sf_socp,
5098                                     sf->sf_sochandle->fcml_portno,
5099                                     fpkt, 1);
5100                 }
5101                 if ((rval == FCML_ABORTED) ||
5102                     (rval == FCML_ABORT_FAILED)) {
5103                     if (abort) {
5104                         pkt->pkt_reason = CMD_ABORTED;
5105                         pkt->pkt_statistics |=
5106                             STAT_ABORTED;
5107                     } else {
5108                         pkt->pkt_reason = CMD_RESET;
5109                         pkt->pkt_statistics |=
5110                             STAT_DEV_RESET;
5111                     }
5112                     cmd->cmd_state = SF_STATE_IDLE;
5113                     if (pkt->pkt_comp)
5114                         (*pkt->pkt_comp)(pkt);
5115                 }
5116                 mutex_enter(&sf->sf_mutex);
5117                 if (lip_cnt != sf->sf_lip_cnt) {
5118                     mutex_exit(&sf->sf_mutex);
5119                     return;
5120                 }
5121                 mutex_exit(&sf->sf_mutex);
5122                 mutex_enter(&target->sft_pkt_mutex);
5123                 cmd = target->sft_pkt_head;
5124             } else {
5125                 mutex_exit(&cmd->cmd_abort_mutex);
5126                 cmd = cmd->cmd_forw;
5127             }
5128         }
5129         mutex_exit(&target->sft_pkt_mutex);
5130     }
5131 }

5134 /*
5135  * called by the transport to start a packet
5136  */
5137 static int
5138 sf_start(struct scsi_address *ap, struct scsi_pkt *pkt)
5139 {
5140     struct sf *sf = ADDR2SF(ap);

```

```

5141 struct sf_target *target = ADDR2TARGET(ap);
5142 struct sf_pkt *cmd = PKT2CMD(pkt);
5143 int rval;

5146 SF_DEBUG(6, (sf, CE_NOTE, "sf_start\n"));

5148 if (cmd->cmd_state == SF_STATE_ISSUED) {
5149     cmn_err(CE_PANIC, "sf: issuing packet twice 0x%p\n",
5150         (void *)cmd);
5151 }

5153 /* prepare the packet for transport */
5154 if ((rval = sf_prepare_pkt(sf, cmd, target)) != TRAN_ACCEPT) {
5155     return (rval);
5156 }

5158 if (target->sft_state & (SF_TARGET_BUSY|SF_TARGET_OFFLINE)) {
5159     if (target->sft_state & SF_TARGET_OFFLINE) {
5160         return (TRAN_FATAL_ERROR);
5161     }
5162     if (pkt->pkt_flags & FLAG_NOINTR) {
5163         return (TRAN_BUSY);
5164     }
5165     mutex_enter(&sf->sf_cmd_mutex);
5166     sf->sf_use_lock = TRUE;
5167     goto enqueue;
5168 }

5171 /* if no interrupts then do polled I/O */
5172 if (pkt->pkt_flags & FLAG_NOINTR) {
5173     return (sf_dopoll(sf, cmd));
5174 }

5176 /* regular interrupt-driven I/O */

5178 if (!sf->sf_use_lock) {

5180     /* locking no needed */

5182     cmd->cmd_timeout = cmd->cmd_pkt->pkt_time ?
5183         sf_watchdog_time + cmd->cmd_pkt->pkt_time : 0;
5184     cmd->cmd_state = SF_STATE_ISSUED;

5186     /* call the transport to send a pkt */
5187     if (soc_transport(sf->sf_sochandle, cmd->cmd_fp_pkt,
5188         FCAL_NOSLEEP, CQ_REQUEST_1) != FCAL_TRANSPORT_SUCCESS) {
5189         cmd->cmd_state = SF_STATE_IDLE;
5190         return (TRAN_BADPKT);
5191     }
5192     return (TRAN_ACCEPT);
5193 }

5195 /* regular I/O using locking */

5197 mutex_enter(&sf->sf_cmd_mutex);
5198 if ((sf->sf_ncmds >= sf->sf_throttle) ||
5199     (sf->sf_pkt_head != NULL)) {
5200 enqueue:
5201     /*
5202     * either we're throttling back or there are already commands
5203     * on the queue, so enqueue this one for later
5204     */
5205     cmd->cmd_flags |= CFLAG_IN_QUEUE;
5206     if (sf->sf_pkt_head != NULL) {

```

```

5207         /* add to the queue */
5208         sf->sf_pkt_tail->cmd_next = cmd;
5209         cmd->cmd_next = NULL;
5210         sf->sf_pkt_tail = cmd;
5211     } else {
5212         /* this is the first entry in the queue */
5213         sf->sf_pkt_head = sf->sf_pkt_tail = cmd;
5214         cmd->cmd_next = NULL;
5215     }
5216     mutex_exit(&sf->sf_cmd_mutex);
5217     return (TRAN_ACCEPT);
5218 }

5220 /*
5221  * start this packet now
5222  */

5224 /* still have cmd mutex */
5225 return (sf_start_internal(sf, cmd));
5226 }

5229 /*
5230  * internal routine to start a packet from the queue now
5231  */
5232 /* enter with cmd mutex held and leave with it released
5233  */
5234 static int
5235 sf_start_internal(struct sf *sf, struct sf_pkt *cmd)
5236 {
5237     /* we have the cmd mutex */
5238     sf->sf_ncmds++;
5239     mutex_exit(&sf->sf_cmd_mutex);

5241     ASSERT(cmd->cmd_state != SF_STATE_ISSUED);
5242     SF_DEBUG(6, (sf, CE_NOTE, "sf_start_internal\n"));

5244     cmd->cmd_timeout = cmd->cmd_pkt->pkt_time ? sf_watchdog_time +
5245         cmd->cmd_pkt->pkt_time : 0;
5246     cmd->cmd_state = SF_STATE_ISSUED;

5248     /* call transport to send the pkt */
5249     if (soc_transport(sf->sf_sochandle, cmd->cmd_fp_pkt, FCAL_NOSLEEP,
5250         CQ_REQUEST_1) != FCAL_TRANSPORT_SUCCESS) {
5251         cmd->cmd_state = SF_STATE_IDLE;
5252         mutex_enter(&sf->sf_cmd_mutex);
5253         sf->sf_ncmds--;
5254         mutex_exit(&sf->sf_cmd_mutex);
5255         return (TRAN_BADPKT);
5256     }
5257     return (TRAN_ACCEPT);
5258 }

5261 /*
5262  * prepare a packet for transport
5263  */
5264 static int
5265 sf_prepare_pkt(struct sf *sf, struct sf_pkt *cmd, struct sf_target *target)
5266 {
5267     struct fcp_cmd *fcmd = cmd->cmd_block;

5269     /* XXXX Need to set the LUN ? */
5270     bcopy((caddr_t)&target->sft_lun.b,
5271         (caddr_t)&fcmd->fcp_ent_addr,
5272         FCP_LUN_SIZE);

```

```

5273 cmd->cmd_pkt->pkt_reason = CMD_CMPLT;
5274 cmd->cmd_pkt->pkt_state = 0;
5275 cmd->cmd_pkt->pkt_statistics = 0;

5278 if ((cmd->cmd_pkt->pkt_comp == NULL) &&
5279     ((cmd->cmd_pkt->pkt_flags & FLAG_NOINTR) == 0)) {
5280     return (TRAN_BADPKT);
5281 }

5283 /* invalidate imp field(s) of rsp block */
5284 cmd->cmd_rsp_block->fcp_u.i_fcp_status = SF_BAD_DMA_MAGIC;

5286 /* set up amt of I/O to do */
5287 if (cmd->cmd_flags & CFLAG_DMAVALID) {
5288     cmd->cmd_pkt->pkt_resid = cmd->cmd_dmacount;
5289     if (cmd->cmd_flags & CFLAG_CMDIOPB) {
5290         (void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,
5291                             DDI_DMA_SYNC_FORDEV);
5292     }
5293 } else {
5294     cmd->cmd_pkt->pkt_resid = 0;
5295 }

5297 /* set up the Tagged Queuing type */
5298 if (cmd->cmd_pkt->pkt_flags & FLAG_HTAG) {
5299     fcmd->fcp_cntl.cntl_qtype = FCP_QTYPE_HEAD_OF_Q;
5300 } else if (cmd->cmd_pkt->pkt_flags & FLAG_OTAG) {
5301     fcmd->fcp_cntl.cntl_qtype = FCP_QTYPE_ORDERED;
5302 }

5304 /*
5305  * Sync the cmd segment
5306  */
5307 (void) ddi_dma_sync(cmd->cmd_cr_pool->cmd_dma_handle,
5308                    (caddr_t)fcmd - cmd->cmd_cr_pool->cmd_base,
5309                    sizeof (struct fcp_cmd), DDI_DMA_SYNC_FORDEV);

5311 sf_fill_ids(sf, cmd, target);
5312 return (TRAN_ACCEPT);
5313 }

5316 /*
5317  * fill in packet hdr source and destination IDs and hdr byte count
5318  */
5319 static void
5320 sf_fill_ids(struct sf *sf, struct sf_pkt *cmd, struct sf_target *target)
5321 {
5322     struct fcal_packet *fpkt = cmd->cmd_fp_pkt;
5323     fc_frame_header_t *hp;

5326 hp = &fpkt->fcal_socal_request.sr_fc_frame_hdr;
5327 hp->d_id = target->sft_al_pa;
5328 hp->s_id = sf->sf_al_pa;
5329 fpkt->fcal_socal_request.sr_soc_hdr.sh_byte_cnt =
5330     cmd->cmd_dmacookie.dmac_size;
5331 }

5334 /*
5335  * do polled I/O using transport
5336  */
5337 static int
5338 sf_dopoll(struct sf *sf, struct sf_pkt *cmd)

```

```

5339 {
5340     int timeout;
5341     int rval;

5344     mutex_enter(&sf->sf_cmd_mutex);
5345     sf->sf_ncmds++;
5346     mutex_exit(&sf->sf_cmd_mutex);

5348     timeout = cmd->cmd_pkt->pkt_time ? cmd->cmd_pkt->pkt_time
5349             : SF_POLL_TIMEOUT;
5350     cmd->cmd_timeout = 0;
5351     cmd->cmd_fp_pkt->fcal_pkt_comp = NULL;
5352     cmd->cmd_state = SF_STATE_ISSUED;

5354     /* call transport to send a pkt polled */
5355     rval = soc_transport_poll(sf->sf_sochandle, cmd->cmd_fp_pkt,
5356                             timeout*1000000, CQ_REQUEST_I);
5357     mutex_enter(&cmd->cmd_abort_mutex);
5358     cmd->cmd_fp_pkt->fcal_pkt_comp = sf_cmd_callback;
5359     if (rval != FCAL_TRANSPORT_SUCCESS) {
5360         if (rval == FCAL_TRANSPORT_TIMEOUT) {
5361             cmd->cmd_state = SF_STATE_ABORTING;
5362             mutex_exit(&cmd->cmd_abort_mutex);
5363             (void) sf_target_timeout(sf, cmd);
5364         } else {
5365             mutex_exit(&cmd->cmd_abort_mutex);
5366         }
5367         cmd->cmd_state = SF_STATE_IDLE;
5368         cmd->cmd_fp_pkt->fcal_pkt_comp = sf_cmd_callback;
5369         mutex_enter(&sf->sf_cmd_mutex);
5370         sf->sf_ncmds--;
5371         mutex_exit(&sf->sf_cmd_mutex);
5372         return (TRAN_BADPKT);
5373     }
5374     mutex_exit(&cmd->cmd_abort_mutex);
5375     cmd->cmd_fp_pkt->fcal_pkt_comp = sf_cmd_callback;
5376     sf_cmd_callback(cmd->cmd_fp_pkt);
5377     return (TRAN_ACCEPT);
5378 }

5381 /* a shortcut for defining debug messages below */
5382 #ifdef DEBUG
5383 #define SF_DMSG1(s)          msg1 = s
5384 #else
5385 #define SF_DMSG1(s)          /* do nothing */
5386 #endif

5389 /*
5390  * the pkt_comp callback for command packets
5391  */
5392 static void
5393 sf_cmd_callback(struct fcal_packet *fpkt)
5394 {
5395     struct sf_pkt *cmd = (struct sf_pkt *)fpkt->fcal_pkt_private;
5396     struct scsi_pkt *pkt = cmd->cmd_pkt;
5397     struct sf *sf = ADDR2SF(&pkt->pkt_address);
5398     struct sf_target *target = ADDR2TARGET(&pkt->pkt_address);
5399     struct fcp_rsp *rsp;
5400     char *msg1 = NULL;
5401     char *msg2 = NULL;
5402     short ncmds;
5403     int tgt_id;
5404     int good_scsi_status = TRUE;

```

```

5408     if (cmd->cmd_state == SF_STATE_IDLE) {
5409         cmn_err(CE_PANIC, "sf: completing idle packet 0x%p\n",
5410             (void *)cmd);
5411     }

5413     mutex_enter(&cmd->cmd_abort_mutex);
5414     if (cmd->cmd_state == SF_STATE_ABORTING) {
5415         /* cmd already being aborted -- nothing to do */
5416         mutex_exit(&cmd->cmd_abort_mutex);
5417         return;
5418     }

5420     cmd->cmd_state = SF_STATE_IDLE;
5421     mutex_exit(&cmd->cmd_abort_mutex);

5423     if (fpkt->fcal_pkt_status == FCAL_STATUS_OK) {

5425         (void) ddi_dma_sync(cmd->cmd_cr_pool->rsp_dma_handle,
5426             (caddr_t)cmd->cmd_rsp_block - cmd->cmd_cr_pool->rsp_base,
5427             FCP_MAX_RSP_IU_SIZE, DDI_DMA_SYNC_FORKERNEL);

5429         rsp = (struct fcp_rsp *)cmd->cmd_rsp_block;

5431         if (rsp->fcp_u.i_fcp_status == SF_BAD_DMA_MAGIC) {

5433             if (sf_core && (sf_core & SF_CORE_BAD_DMA)) {
5434                 sf_token = (int *) (uintptr_t)
5435                     fpkt->fcal_socal_request.\
5436                     sr_soc_hdr.sh_request_token;
5437                 (void) soc_take_core(sf->sf_sochandle,
5438                     sf->sf_socp);
5439             }

5441             pkt->pkt_reason = CMD_INCOMPLETE;
5442             pkt->pkt_state = STATE_GOT_BUS;
5443             pkt->pkt_statistics |= STAT_ABORTED;

5445         } else {

5447             pkt->pkt_state = STATE_GOT_BUS | STATE_GOT_TARGET |
5448                 STATE_SENT_CMD | STATE_GOT_STATUS;
5449             pkt->pkt_resid = 0;
5450             if (cmd->cmd_flags & CFLAG_DMAVALID) {
5451                 pkt->pkt_state |= STATE_XFERRED_DATA;
5452             }

5454             if ((pkt->pkt_scbp != NULL) &&
5455                 ((* (pkt->pkt_scbp) =
5456                     rsp->fcp_u.fcp_status.scsi_status)
5457                 != STATUS_GOOD)) {
5458                 good_scsi_status = FALSE;
5459             }
5460             /*
5461              * The next two checks make sure that if there
5462              * is no sense data or a valid response and
5463              * the command came back with check condition,
5464              * the command should be retried
5465              */
5466             if (!rsp->fcp_u.fcp_status.rsp_len_set &&
5467                 !rsp->fcp_u.fcp_status.sense_len_set) {
5468                 pkt->pkt_state &= ~STATE_XFERRED_DATA;
5469                 pkt->pkt_resid = cmd->cmd_dmacount;
5470             }

```

```

5472     if ((cmd->cmd_flags & CFLAG_CMDIOPB) &&
5473         (pkt->pkt_state & STATE_XFERRED_DATA)) {
5474         (void) ddi_dma_sync(cmd->cmd_dmahandle, 0,
5475             (uint_t)0, DDI_DMA_SYNC_FORCFPU);
5476     }
5477     /*
5478      * Update the transfer resid, if appropriate
5479      */
5480     if (rsp->fcp_u.fcp_status.resid_over ||
5481         rsp->fcp_u.fcp_status.resid_under)
5482         pkt->pkt_resid = rsp->fcp_resid;

5484     /*
5485      * Check to see if the SCSI command failed.
5486      */
5487     /*

5489     /*
5490      * First see if we got a FCP protocol error.
5491      */
5492     if (rsp->fcp_u.fcp_status.rsp_len_set) {
5493         struct fcp_rsp_info *bep;

5495         bep = (struct fcp_rsp_info *)
5496             (&rsp->fcp_response_len + 1);
5497         if (bep->rsp_code != FCP_NO_FAILURE) {
5498             pkt->pkt_reason = CMD_TRAN_ERR;
5499             tgt_id = pkt->pkt_address.a_target;
5500             switch (bep->rsp_code) {
5501                 case FCP_CMND_INVALID:
5502                     SF_DMSG1("FCP_RSP FCP_CMND "
5503                         "fields invalid");
5504                     break;
5505                 case FCP_TASK_MGMT_NOT_SUPPTD:
5506                     SF_DMSG1("FCP_RSP Task "
5507                         "Management Function "
5508                         "Not Supported");
5509                     break;
5510                 case FCP_TASK_MGMT_FAILED:
5511                     SF_DMSG1("FCP_RSP Task "
5512                         "Management Function "
5513                         "Failed");
5514                     sf->sf_stats.tstats[tgt_id].
5515                         task_mgmt_failures++;
5516                     break;
5517                 case FCP_DATA_RO_MISMATCH:
5518                     SF_DMSG1("FCP_RSP FCP_DATA RO "
5519                         "mismatch with "
5520                         "FCP_XFER_RDY DATA_RO");
5521                     sf->sf_stats.tstats[tgt_id].
5522                         data_ro_mismatches++;
5523                     break;
5524                 case FCP_DL_LEN_MISMATCH:
5525                     SF_DMSG1("FCP_RSP FCP_DATA "
5526                         "length "
5527                         "different than BURST_LEN");
5528                     sf->sf_stats.tstats[tgt_id].
5529                         dl_len_mismatches++;
5530                     break;
5531             }
5532             default:
5533                 SF_DMSG1("FCP_RSP invalid "
5534                     "RSP_CODE");
5535                 break;
5536         }

```

```

5537     }
5539     /*
5540     * See if we got a SCSI error with sense data
5541     */
5542     if (rsp->fcpl_u.fcp_status.sense_len_set) {
5543         uchar_t rqlen = min(rsp->fcpl_sense_len,
5544             sizeof (struct scsi_extended_sense));
5545         caddr_t sense = (caddr_t)rsp +
5546             sizeof (struct fcp_rsp) +
5547             rsp->fcpl_response_len;
5548         struct scsi_arq_status *arq;
5549         struct scsi_extended_sense *sensep =
5550             (struct scsi_extended_sense *)sense;
5552
5553         if (rsp->fcpl_u.fcp_status.scsl_status !=
5554             STATUS_GOOD) {
5555             if (rsp->fcpl_u.fcp_status.scsl_status
5556                 == STATUS_CHECK) {
5557                 if (sensep->es_key ==
5558                     KEY_RECOVERABLE_ERROR)
5559                     good_scsl_status = 1;
5560                 if (sensep->es_key ==
5561                     KEY_UNIT_ATTENTION &&
5562                     sensep->es_add_code == 0x3f &&
5563                     sensep->es_qual_code == 0x0e) {
5564                     /* REPORT LUNS HAS CHANGED */
5565                     sf_log(sf, CE_NOTE,
5566                         "!REPORT LUNS HAS CHANGED\n");
5567                     sf_force_lip(sf);
5568                 }
5569             }
5571
5572             if ((pkt->pkt_scbp != NULL) &&
5573                 (cmd->cmd_scbllen >=
5574                     sizeof (struct scsi_arq_status))) {
5575
5576                 pkt->pkt_state |= STATE_ARQ_DONE;
5577
5578                 arq = (struct scsi_arq_status *)pkt->pkt_scbp;
5579                 /*
5580                 * copy out sense information
5581                 */
5582                 bcopy(sense, (caddr_t)&arq->sts_sensedata,
5583                     rqlen);
5584                 arq->sts_rqpkt_resid =
5585                     sizeof (struct scsi_extended_sense) -
5586                     rqlen;
5587                 *((uchar_t *)&arq->sts_rqpkt_status) =
5588                     STATUS_GOOD;
5589                 arq->sts_rqpkt_reason = 0;
5590                 arq->sts_rqpkt_statistics = 0;
5591                 arq->sts_rqpkt_state = STATE_GOT_BUS |
5592                     STATE_GOT_TARGET | STATE_SENT_CMD |
5593                     STATE_GOT_STATUS | STATE_ARQ_DONE |
5594                     STATE_XFERRED_DATA;
5595                 target->sft_alive = TRUE;
5596             }
5598         /*
5599         * The firmware returns the number of bytes actually
5600         * xfered into/out of host. Compare this with what
5601         * we asked and if it is different, we lost frames ?
5602         */

```

```

5603         if ((pkt->pkt_reason == 0) && (pkt->pkt_resid == 0) &&
5604             (good_scsl_status) &&
5605             (pkt->pkt_state & STATE_XFERRED_DATA) &&
5606             (!(cmd->cmd_flags & CFLAG_CMDIOPE)) &&
5607             (target->sft_device_type != DTYPE_ESI)) {
5608             int byte_cnt =
5609                 fpkt->fcpl_socal_request.
5610                 sr_soc_hdr.sh_byte_cnt;
5611             if (cmd->cmd_flags & CFLAG_DMASEND) {
5612                 if (byte_cnt != 0) {
5613                     sf_log(sf, CE_NOTE,
5614                         "!sf_cmd_callback: Lost Frame: "
5615                         "(write) received 0x%x expected "
5616                         "0x%x target 0x%x\n",
5617                         byte_cnt, cmd->cmd_dmacount,
5618                         sf_alpa_to_switch[
5619                             target->sft_al_pa]);
5620                     pkt->pkt_reason = CMD_INCOMPLETE;
5621                     pkt->pkt_statistics |= STAT_ABORTED;
5622                 }
5623             } else if (byte_cnt < cmd->cmd_dmacount) {
5624                 sf_log(sf, CE_NOTE,
5625                     "!sf_cmd_callback: "
5626                     "Lost Frame: (read) "
5627                     "received 0x%x expected 0x%x "
5628                     "target 0x%x\n", byte_cnt,
5629                     cmd->cmd_dmacount,
5630                     sf_alpa_to_switch[
5631                         target->sft_al_pa]);
5632                 pkt->pkt_reason = CMD_INCOMPLETE;
5633                 pkt->pkt_statistics |= STAT_ABORTED;
5634             }
5635         }
5636     }
5638     } else {
5640         /* pkt status was not ok */
5642         switch (fpkt->fcpl_pkt_status) {
5644             case FCAL_STATUS_ERR_OFFLINE:
5645                 SF_DMSG1("Fibre Channel Offline");
5646                 mutex_enter(&target->sft_mutex);
5647                 if (!(target->sft_state & SF_TARGET_OFFLINE)) {
5648                     target->sft_state |= (SF_TARGET_BUSY
5649                         | SF_TARGET_MARK);
5650                 }
5651                 mutex_exit(&target->sft_mutex);
5652                 (void) ndi_event_retrieve_cookie(sf->sft_event_hdl,
5653                     target->sft_dip, FCAL_REMOVE_EVENT,
5654                     &sf_remove_eid, NDI_EVENT_NOPASS);
5655                 (void) ndi_event_run_callbacks(sf->sft_event_hdl,
5656                     target->sft_dip, sf_remove_eid, NULL);
5657                 pkt->pkt_reason = CMD_TRAN_ERR;
5658                 pkt->pkt_statistics |= STAT_BUS_RESET;
5659                 break;
5661             case FCAL_STATUS_MAX_XCHG_EXCEEDED:
5662                 sf_throttle(sf);
5663                 sf->sft_use_lock = TRUE;
5664                 pkt->pkt_reason = CMD_TRAN_ERR;
5665                 pkt->pkt_state = STATE_GOT_BUS;
5666                 pkt->pkt_statistics |= STAT_ABORTED;
5667                 break;

```

```

5669     case FCAL_STATUS_TIMEOUT:
5670         SF_DMSG1("Fibre Channel Timeout");
5671         pkt->pkt_reason = CMD_TIMEOUT;
5672         break;

5674     case FCAL_STATUS_ERR_OVERRUN:
5675         SF_DMSG1("CMD_DATA_OVR");
5676         pkt->pkt_reason = CMD_DATA_OVR;
5677         break;

5679     case FCAL_STATUS_UNKNOWN_CQ_TYPE:
5680         SF_DMSG1("Unknown CQ type");
5681         pkt->pkt_reason = CMD_TRAN_ERR;
5682         break;

5684     case FCAL_STATUS_BAD_SEG_CNT:
5685         SF_DMSG1("Bad SEG CNT");
5686         pkt->pkt_reason = CMD_TRAN_ERR;
5687         break;

5689     case FCAL_STATUS_BAD_XID:
5690         SF_DMSG1("Fibre Channel Invalid X_ID");
5691         pkt->pkt_reason = CMD_TRAN_ERR;
5692         break;

5694     case FCAL_STATUS_XCHG_BUSY:
5695         SF_DMSG1("Fibre Channel Exchange Busy");
5696         pkt->pkt_reason = CMD_TRAN_ERR;
5697         break;

5699     case FCAL_STATUS_INSUFFICIENT_CQES:
5700         SF_DMSG1("Insufficient CQEs");
5701         pkt->pkt_reason = CMD_TRAN_ERR;
5702         break;

5704     case FCAL_STATUS_ALLOC_FAIL:
5705         SF_DMSG1("ALLOC FAIL");
5706         pkt->pkt_reason = CMD_TRAN_ERR;
5707         break;

5709     case FCAL_STATUS_BAD_SID:
5710         SF_DMSG1("Fibre Channel Invalid S_ID");
5711         pkt->pkt_reason = CMD_TRAN_ERR;
5712         break;

5714     case FCAL_STATUS_INCOMPLETE_DMA_ERR:
5715         if (sf_core && (sf_core & SF_CORE_INCOMPLETE_DMA)) {
5716             sf_token = (int *) (uintptr_t)
5717                 fpkt->fcal_socall_request.\
5718                 sr_soc_hdr.sh_request_token;
5719             (void) soc_take_core(sf->sf_sochandle,
5720                 sf->sf_socp);
5721             sf_core = 0;
5722         }
5723         msg2 =
5724             "INCOMPLETE DMA XFER due to bad SOC+ card, replace HBA";
5725         pkt->pkt_reason = CMD_INCOMPLETE;
5726         pkt->pkt_state = STATE_GOT_BUS;
5727         pkt->pkt_statistics |= STAT_ABORTED;
5728         break;

5730     case FCAL_STATUS_CRC_ERR:
5731         msg2 = "Fibre Channel CRC Error on frames";
5732         pkt->pkt_reason = CMD_INCOMPLETE;
5733         pkt->pkt_state = STATE_GOT_BUS;
5734         pkt->pkt_statistics |= STAT_ABORTED;

```

```

5735         break;

5737     case FCAL_STATUS_NO_SEQ_INIT:
5738         SF_DMSG1("Fibre Channel Seq Init Error");
5739         pkt->pkt_reason = CMD_TRAN_ERR;
5740         break;

5742     case FCAL_STATUS_OPEN_FAIL:
5743         pkt->pkt_reason = CMD_TRAN_ERR;
5744         SF_DMSG1("Fibre Channel Open Failure");
5745         if ((target->sft_state & (SF_TARGET_BUSY |
5746             SF_TARGET_MARK | SF_TARGET_OFFLINE)) == 0) {
5747             sf_log(sf, CE_NOTE,
5748                 "!Open failure to target 0x%x "
5749                 "forcing LIP\n",
5750                 sf_alpa_to_switch[target->sft_al_pa]);
5751             sf_force_lip(sf);
5752         }
5753         break;

5756     case FCAL_STATUS_ONLINE_TIMEOUT:
5757         SF_DMSG1("Fibre Channel Online Timeout");
5758         pkt->pkt_reason = CMD_TRAN_ERR;
5759         break;

5761     default:
5762         SF_DMSG1("Unknown FC Status");
5763         pkt->pkt_reason = CMD_TRAN_ERR;
5764         break;
5765     }
5766 }

5768 #ifdef DEBUG
5769 /*
5770  * msg1 will be non-NULL if we've detected some sort of error
5771  */
5772 if (msg1 != NULL && sfdebug >= 4) {
5773     sf_log(sf, CE_WARN,
5774         "!Transport error on cmd=0x%p target=0x%x: %s\n",
5775         (void *)fpkt, pkt->pkt_address.a_target, msg1);
5776 }
5777 #endif

5779 if (msg2 != NULL) {
5780     sf_log(sf, CE_WARN, "!Transport error on target=0x%x: %s\n",
5781         pkt->pkt_address.a_target, msg2);
5782 }

5784 ncmds = fpkt->fcal_ncmds;
5785 ASSERT(ncmds >= 0);
5786 if (ncmds >= (sf->sf_throttle - SF_HI_CMD_DELTA)) {
5787 #ifdef DEBUG
5788     if (!sf->sf_use_lock) {
5789         SF_DEBUG(4, (sf, CE_NOTE, "use lock flag on\n"));
5790     }
5791 #endif
5792     sf->sf_use_lock = TRUE;
5793 }

5795 mutex_enter(&sf->sf_cmd_mutex);
5796 sf->sf_ncmds = ncmds;
5797 sf_throttle_start(sf);
5798 mutex_exit(&sf->sf_cmd_mutex);

5800 if (!msg1 && !msg2)

```

```

5801         SF_DEBUG(6, (sf, CE_NOTE, "Completing pkt 0x%p\n",
5802             (void *)pkt));
5803     if (pkt->pkt_comp != NULL) {
5804         (*pkt->pkt_comp)(pkt);
5805     }
5806 }

5808 #undef SF_DMSG1

5812 /*
5813  * start throttling for this instance
5814  */
5815 static void
5816 sf_throttle_start(struct sf *sf)
5817 {
5818     struct sf_pkt *cmd, *prev_cmd = NULL;
5819     struct scsi_pkt *pkt;
5820     struct sf_target *target;

5823     ASSERT(mutex_owned(&sf->sf_cmd_mutex));

5825     cmd = sf->sf_pkt_head;
5826     while ((cmd != NULL) &&
5827         (sf->sf_state == SF_STATE_ONLINE) &&
5828         (sf->sf_ncmds < sf->sf_throttle)) {

5830         pkt = CMD2PKT(cmd);

5832         target = ADDR2TARGET(&pkt->pkt_address);
5833         if (target->sft_state & SF_TARGET_BUSY) {
5834             /* this command is busy -- go to next */
5835             ASSERT(cmd->cmd_state != SF_STATE_ISSUED);
5836             prev_cmd = cmd;
5837             cmd = cmd->cmd_next;
5838             continue;
5839         }

5841         ASSERT(cmd->cmd_state != SF_STATE_ISSUED);

5843         /* this cmd not busy and not issued */

5845         /* remove this packet from the queue */
5846         if (sf->sf_pkt_head == cmd) {
5847             /* this was the first packet */
5848             sf->sf_pkt_head = cmd->cmd_next;
5849         } else if (sf->sf_pkt_tail == cmd) {
5850             /* this was the last packet */
5851             sf->sf_pkt_tail = prev_cmd;
5852             if (prev_cmd != NULL) {
5853                 prev_cmd->cmd_next = NULL;
5854             }
5855         } else {
5856             /* some packet in the middle of the queue */
5857             ASSERT(prev_cmd != NULL);
5858             prev_cmd->cmd_next = cmd->cmd_next;
5859         }
5860         cmd->cmd_flags &= ~CFLAG_IN_QUEUE;

5862         if (target->sft_state & SF_TARGET_OFFLINE) {
5863             mutex_exit(&sf->sf_cmd_mutex);
5864             pkt->pkt_reason = CMD_TRAN_ERR;
5865             if (pkt->pkt_comp != NULL) {
5866                 (*pkt->pkt_comp)(cmd->cmd_pkt);

```

```

5867     } else {
5868         sf_fill_ids(sf, cmd, target);
5869         if (sf_start_internal(sf, cmd) != TRAN_ACCEPT) {
5870             pkt->pkt_reason = CMD_TRAN_ERR;
5871             if (pkt->pkt_comp != NULL) {
5872                 (*pkt->pkt_comp)(cmd->cmd_pkt);
5873             }
5874         }
5875     }
5876 }
5877 mutex_enter(&sf->sf_cmd_mutex);
5878 cmd = sf->sf_pkt_head;
5879 prev_cmd = NULL;
5880 }
5881 }

5884 /*
5885  * called when the max exchange value is exceeded to throttle back commands
5886  */
5887 static void
5888 sf_throttle(struct sf *sf)
5889 {
5890     int cmdmax = sf->sf_sochandle->fcml_cmdmax;

5893     mutex_enter(&sf->sf_cmd_mutex);

5895     sf->sf_flag = TRUE;

5897     if (sf->sf_ncmds > (cmdmax / 2)) {
5898         sf->sf_throttle = cmdmax / 2;
5899     } else {
5900         if (sf->sf_ncmds > SF_DECR_DELTA) {
5901             sf->sf_throttle = sf->sf_ncmds - SF_DECR_DELTA;
5902         } else {
5903             /*
5904              * This case is just a safeguard, should not really
5905              * happen(ncmds < SF_DECR_DELTA and MAX_EXCHG exceed
5906              */
5907             sf->sf_throttle = SF_DECR_DELTA;
5908         }
5909     }
5910     mutex_exit(&sf->sf_cmd_mutex);

5912     sf = sf->sf_sibling;
5913     if (sf != NULL) {
5914         mutex_enter(&sf->sf_cmd_mutex);
5915         sf->sf_flag = TRUE;
5916         if (sf->sf_ncmds >= (cmdmax / 2)) {
5917             sf->sf_throttle = cmdmax / 2;
5918         } else {
5919             if (sf->sf_ncmds > SF_DECR_DELTA) {
5920                 sf->sf_throttle = sf->sf_ncmds - SF_DECR_DELTA;
5921             } else {
5922                 sf->sf_throttle = SF_DECR_DELTA;
5923             }
5924         }

5926         mutex_exit(&sf->sf_cmd_mutex);
5927     }
5928 }

5931 /*
5932  * sf watchdog routine, called for a timeout

```



```

5933 */
5934 /*ARGSUSED*/
5935 static void
5936 sf_watch(void *arg)
5937 {
5938     struct sf *sf;
5939     struct sf_els_hdr *privp;
5940     static int count = 0, pscan_count = 0;
5941     int cmdmax, i, mescount = 0;
5942     struct sf_target *target;

5945     sf_watchdog_time += sf_watchdog_timeout;
5946     count++;
5947     pscan_count++;

5949     mutex_enter(&sf_global_mutex);
5950     sf_watch_running = 1;
5951     for (sf = sf_head; sf != NULL; sf = sf->sf_next) {

5953         mutex_exit(&sf_global_mutex);

5955         /* disable throttling while we're suspended */
5956         mutex_enter(&sf->sf_mutex);
5957         if (sf->sf_state & SF_STATE_SUSPENDED) {
5958             mutex_exit(&sf->sf_mutex);
5959             SF_DEBUG(1, (sf, CE_CONT,
5960                 "sf_watch, sf%d:throttle disabled "
5961                 "due to DDI_SUSPEND\n",
5962                 ddi_get_instance(sf->sf_dip)));
5963             mutex_enter(&sf_global_mutex);
5964             continue;
5965         }
5966         mutex_exit(&sf->sf_mutex);

5968         cmdmax = sf->sf_sochandle->fcal_cmdmax;

5970         if (sf->sf_take_core) {
5971             (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
5972         }

5974         mutex_enter(&sf->sf_cmd_mutex);

5976         if (!sf->sf_flag) {
5977             if (sf->sf_throttle < (cmdmax / 2)) {
5978                 sf->sf_throttle = cmdmax / 2;
5979             } else if ((sf->sf_throttle += SF_INCR_DELTA) >
5980                 cmdmax) {
5981                 sf->sf_throttle = cmdmax;
5982             }
5983         } else {
5984             sf->sf_flag = FALSE;
5985         }

5987         sf->sf_ncmds_exp_avg = (sf->sf_ncmds + sf->sf_ncmds_exp_avg)
5988             >> 2;
5989         if ((sf->sf_ncmds <= (sf->sf_throttle - SF_LO_CMD_DELTA)) &&
5990             (sf->sf_pkt_head == NULL)) {
5991             #ifdef DEBUG
5992                 if (sf->sf_use_lock) {
5993                     SF_DEBUG(4, (sf, CE_NOTE,
5994                         "use lock flag off\n"));
5995                 }
5996             #endif
5997             sf->sf_use_lock = FALSE;
5998         }

```

```

6000         if (sf->sf_state == SF_STATE_ONLINE && sf->sf_pkt_head &&
6001             sf->sf_ncmds < sf->sf_throttle) {
6002             sf_throttle_start(sf);
6003         }

6005         mutex_exit(&sf->sf_cmd_mutex);

6007         if (pscan_count >= sf_pool_scan_cnt) {
6008             if (sf->sf_ncmds_exp_avg < (sf->sf_cr_pool_cnt <<
6009                 SF_LOG2_ELEMS_IN_POOL) - SF_FREE_CR_EPSILON) {
6010                 sf_crpool_free(sf);
6011             }
6012         }
6013         mutex_enter(&sf->sf_mutex);

6015         privp = sf->sf_els_list;
6016         while (privp != NULL) {
6017             if (privp->timeout < sf_watchdog_time) {
6018                 /* timeout this command */
6019                 privp = sf_els_timeout(sf, privp);
6020             } else if ((privp->timeout == SF_INVALID_TIMEOUT) &&
6021                 (privp->lip_cnt != sf->sf_lip_cnt)) {
6022                 if (privp->prev != NULL) {
6023                     privp->prev->next = privp->next;
6024                 }
6025                 if (sf->sf_els_list == privp) {
6026                     sf->sf_els_list = privp->next;
6027                 }
6028                 if (privp->next != NULL) {
6029                     privp->next->prev = privp->prev;
6030                 }
6031                 mutex_exit(&sf->sf_mutex);
6032                 sf_els_free(privp->fpkt);
6033                 mutex_enter(&sf->sf_mutex);
6034                 privp = sf->sf_els_list;
6035             } else {
6036                 privp = privp->next;
6037             }
6038         }

6040         if (sf->sf_online_timer && sf->sf_online_timer <
6041             sf_watchdog_time) {
6042             for (i = 0; i < sf_max_targets; i++) {
6043                 target = sf->sf_targets[i];
6044                 if (target != NULL) {
6045                     if (!mescount && target->sft_state &
6046                         SF_TARGET_BUSY) {
6047                         sf_log(sf, CE_WARN, "!Loop "
6048                             "Unstable: Failed to bring "
6049                             "Loop Online\n");
6050                         mescount = 1;
6051                     }
6052                     target->sft_state |= SF_TARGET_MARK;
6053                 }
6054             }
6055             sf_finish_init(sf, sf->sf_lip_cnt);
6056             sf->sf_state = SF_STATE_INIT;
6057             sf->sf_online_timer = 0;
6058         }

6060         if (sf->sf_state == SF_STATE_ONLINE) {
6061             mutex_exit(&sf->sf_mutex);
6062             if (count >= sf_pkt_scan_cnt) {
6063                 sf_check_targets(sf);
6064             }

```

```

6065     } else if ((sf->sf_state == SF_STATE_OFFLINE) &&
6066                (sf->sf_timer < sf_watchdog_time)) {
6067         for (i = 0; i < sf_max_targets; i++) {
6068             target = sf->sf_targets[i];
6069             if ((target != NULL) &&
6070                 (target->sft_state &
6071                  SF_TARGET_BUSY)) {
6072                 sf_log(sf, CE_WARN,
6073                        "!Offline Timeout\n");
6074                 if (sf_core && (sf_core &
6075                                SF_CORE_OFFLINE_TIMEOUT)) {
6076                     (void) soc_take_core(
6077                         sf->sf_sochandle,
6078                         sf->sf_socp);
6079                     sf_core = 0;
6080                 }
6081                 break;
6082             }
6083         }
6084         sf_finish_init(sf, sf->sf_lip_cnt);
6085         sf->sf_state = SF_STATE_INIT;
6086         mutex_exit(&sf->sf_mutex);
6087     } else {
6088         mutex_exit(&sf->sf_mutex);
6089     }
6090     mutex_enter(&sf_global_mutex);
6091 }
6092 mutex_exit(&sf_global_mutex);
6093 if (count >= sf_pkt_scan_cnt) {
6094     count = 0;
6095 }
6096 if (pscan_count >= sf_pool_scan_cnt) {
6097     pscan_count = 0;
6098 }
6099
6100 /* reset timeout */
6101 sf_watchdog_id = timeout(sf_watch, (caddr_t)0, sf_watchdog_tick);
6102
6103 /* signal waiting thread */
6104 mutex_enter(&sf_global_mutex);
6105 sf_watch_running = 0;
6106 cv_broadcast(&sf_watch_cv);
6107 mutex_exit(&sf_global_mutex);
6108 }
6109
6111 /*
6112  * called during a timeout to check targets
6113  */
6114 static void
6115 sf_check_targets(struct sf *sf)
6116 {
6117     struct sf_target *target;
6118     int i;
6119     struct sf_pkt *cmd;
6120     struct scsi_pkt *pkt;
6121     int lip_cnt;
6122
6123     mutex_enter(&sf->sf_mutex);
6124     lip_cnt = sf->sf_lip_cnt;
6125     mutex_exit(&sf->sf_mutex);
6126
6127     /* check scan all possible targets */
6128     for (i = 0; i < sf_max_targets; i++) {
6129         target = sf->sf_targets[i];
6130         while (target != NULL) {

```

```

6131         mutex_enter(&target->sft_pkt_mutex);
6132         if (target->sft_alive && target->sft_scan_count !=
6133             sf_target_scan_cnt) {
6134             target->sft_alive = 0;
6135             target->sft_scan_count++;
6136             mutex_exit(&target->sft_pkt_mutex);
6137             return;
6138         }
6139         target->sft_alive = 0;
6140         target->sft_scan_count = 0;
6141         cmd = target->sft_pkt_head;
6142         while (cmd != (struct sf_pkt *)&target->sft_pkt_head) {
6143             mutex_enter(&cmd->cmd_abort_mutex);
6144             if (cmd->cmd_state == SF_STATE_ISSUED &&
6145                 ((cmd->cmd_timeout && sf_watchdog_time >
6146                  #ifdef DEBUG
6147                      cmd->cmd_timeout) || sf_abort_flag)) {
6148                 sf_abort_flag = 0;
6149             }
6150             #else
6151             #endif
6152                 cmd->cmd_timeout))) {
6153             cmd->cmd_timeout = 0;
6154             /* prevent reset from getting at this packet */
6155             cmd->cmd_state = SF_STATE_ABORTING;
6156             mutex_exit(&cmd->cmd_abort_mutex);
6157             mutex_exit(&target->sft_pkt_mutex);
6158             sf->sf_stats.tstats[i].timeouts++;
6159             if (sf_target_timeout(sf, cmd))
6160                 return;
6161             else {
6162                 if (lip_cnt != sf->sf_lip_cnt) {
6163                     return;
6164                 } else {
6165                     mutex_enter(&target->
6166                                 sft_pkt_mutex);
6167                     cmd = target->
6168                         sft_pkt_head;
6169                 }
6170             }
6171         }
6172     }
6173     /*
6174     * if the abort and lip fail, a reset will be carried out.
6175     * But the reset will ignore this packet. We have waited at least
6176     * 20 seconds after the initial timeout. Now, complete it here.
6177     * This also takes care of spurious bad aborts.
6178     */
6179     } else if ((cmd->cmd_state ==
6180                SF_STATE_ABORTING) && (cmd->cmd_timeout
6181                <= sf_watchdog_time)) {
6182         cmd->cmd_state = SF_STATE_IDLE;
6183         mutex_exit(&cmd->cmd_abort_mutex);
6184         mutex_exit(&target->sft_pkt_mutex);
6185         SF_DEBUG(1, (sf, CE_NOTE,
6186                    "Command 0x%p to sft 0x%p"
6187                    " delayed release\n",
6188                    (void *)cmd, (void *)target));
6189         pkt = cmd->cmd_pkt;
6190         pkt->pkt_statistics |=
6191             (STAT_TIMEOUT|STAT_ABORTED);
6192         pkt->pkt_reason = CMD_TIMEOUT;
6193         if (pkt->pkt_comp) {
6194             scsi_hba_pkt_comp(pkt);
6195             /* handle deferred_destroy case */
6196         } else {
6197             if ((cmd->cmd_block->fcpl_cntl.
6198                 cntl_reset == 1) ||
6199                 (cmd->cmd_block->

```

```

6197         fcp_cntl.cntl_abort_tsk ==
6198         1)) {
6199             cmd->cmd_block->
6200             fcp_cntl.
6201             cntl_reset = 0;
6202             cmd->cmd_block->
6203             fcp_cntl.
6204             cntl_abort_tsk = 0;
6205             cmd->cmd_fp_pkt->
6206             fcal_pkt_comp =
6207             sf_cmd_callback;
6208             /* for cache */
6209             sf_scsi_destroy_pkt
6210             (&pkt->pkt_address,
6211             pkt);
6212         }
6213     }
6214     mutex_enter(&target->sft_pkt_mutex);
6215     cmd = target->sft_pkt_head;
6216 } else {
6217     mutex_exit(&cmd->cmd_abort_mutex);
6218     cmd = cmd->cmd_forw;
6219 }
6220 }
6221 mutex_exit(&target->sft_pkt_mutex);
6222 target = target->sft_next_lun;
6223 }
6224 }
6225 }

6228 /*
6229  * a command to a target has timed out
6230  * return TRUE iff cmd abort failed or timed out, else return FALSE
6231  */
6232 static int
6233 sf_target_timeout(struct sf *sf, struct sf_pkt *cmd)
6234 {
6235     int rval;
6236     struct scsi_pkt *pkt;
6237     struct fcal_packet *fpkt;
6238     int tgt_id;
6239     int retval = FALSE;

6242     SF_DEBUG(1, (sf, CE_NOTE, "Command 0x%p to target %x timed out\n",
6243     (void *)cmd->cmd_fp_pkt, cmd->cmd_pkt->pkt_address.a_target));

6245     fpkt = cmd->cmd_fp_pkt;

6247     if (sf_core && (sf_core & SF_CORE_CMD_TIMEOUT)) {
6248         sf_token = (int *) (uintptr_t)
6249         fpkt->fcal_social_request.sr_soc_hdr.\
6250         sh_request_token;
6251         (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
6252         sf_core = 0;
6253     }

6255     /* call the transport to abort a command */
6256     rval = soc_abort(sf->sf_sochandle, sf->sf_socp,
6257     sf->sf_sochandle->fcal_portno, fpkt, 1);

6259     switch (rval) {
6260     case FCAL_ABORTED:
6261         SF_DEBUG(1, (sf, CE_NOTE, "Command Abort succeeded\n"));
6262         pkt = cmd->cmd_pkt;

```

```

6263         cmd->cmd_state = SF_STATE_IDLE;
6264         pkt->pkt_statistics |= (STAT_TIMEOUT|STAT_ABORTED);
6265         pkt->pkt_reason = CMD_TIMEOUT;
6266         if (pkt->pkt_comp != NULL) {
6267             (*pkt->pkt_comp)(pkt);
6268         }
6269         break; /* success */

6271     case FCAL_ABORT_FAILED:
6272         SF_DEBUG(1, (sf, CE_NOTE, "Command Abort failed at target\n"));
6273         pkt = cmd->cmd_pkt;
6274         cmd->cmd_state = SF_STATE_IDLE;
6275         pkt->pkt_reason = CMD_TIMEOUT;
6276         pkt->pkt_statistics |= STAT_TIMEOUT;
6277         tgt_id = pkt->pkt_address.a_target;
6278         sf->sf_stats.tstats[tgt_id].abts_failures++;
6279         if (pkt->pkt_comp != NULL) {
6280             (*pkt->pkt_comp)(pkt);
6281         }
6282         break;

6284     case FCAL_BAD_ABORT:
6285         if (sf_core && (sf_core & SF_CORE_BAD_ABORT)) {
6286             sf_token = (int *) (uintptr_t) fpkt->fcal_social_request.\
6287             sr_soc_hdr.sh_request_token;
6288             (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
6289             sf_core = 0;
6290         }
6291         SF_DEBUG(1, (sf, CE_NOTE, "Command Abort bad abort\n"));
6292         cmd->cmd_timeout = sf_watchdog_time + cmd->cmd_pkt->pkt_time
6293         + 20;
6294         break;

6296     case FCAL_TIMEOUT:
6297         retval = TRUE;
6298         break;

6300     default:
6301         pkt = cmd->cmd_pkt;
6302         tgt_id = pkt->pkt_address.a_target;
6303         sf_log(sf, CE_WARN,
6304         "Command Abort failed target 0x%x, forcing a LIP\n", tgt_id);
6305         if (sf_core && (sf_core & SF_CORE_ABORT_TIMEOUT)) {
6306             sf_token = (int *) (uintptr_t) fpkt->fcal_social_request.\
6307             sr_soc_hdr.sh_request_token;
6308             (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
6309             sf_core = 0;
6310         }
6311         sf_force_lip(sf);
6312         retval = TRUE;
6313         break;
6314     }

6316     return (retval);
6317 }

6320 /*
6321  * an ELS command has timed out
6322  * return ???
6323  */
6324 static struct sf_els_hdr *
6325 sf_els_timeout(struct sf *sf, struct sf_els_hdr *privp)
6326 {
6327     struct fcal_packet *fpkt;
6328     int rval, dflag, timeout = SF_ELS_TIMEOUT;

```

```

6329     uint_t lip_cnt = privp->lip_cnt;
6330     uchar_t els_code = privp->els_code;
6331     struct sf_target *target = privp->target;
6332     char what[64];

6334     fpkt = privp->fpkt;
6335     dflag = privp->delayed_retry;
6336     /* use as temporary state variable */
6337     privp->timeout = SF_INVALID_TIMEOUT;
6338     mutex_exit(&sf->sf_mutex);

6340     if (privp->fpkt->fcal_pkt_comp == sf_els_callback) {
6341         /*
6342          * take social core if required. Timeouts for IB and hosts
6343          * are not very interesting, so we take social core only
6344          * if the timeout is *not* for a IB or host.
6345          */
6346         if (sf_core && (sf_core & SF_CORE_ELS_TIMEOUT) &&
6347             ((sf_alpa_to_switch[privp->dest_nport_id] &
6348              0x0d) != 0x0d) && ((privp->dest_nport_id != 1) ||
6349              (privp->dest_nport_id != 2) ||
6350              (privp->dest_nport_id != 4) ||
6351              (privp->dest_nport_id != 8) ||
6352              (privp->dest_nport_id != 0xf))) {
6353             sf_token = (int *) (uintptr_t) fpkt->fcal_social_request.\
6354                 sr_soc_hdr.sh_request_token;
6355             (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
6356             sf_core = 0;
6357         }
6358         (void) sprintf(what, "ELS 0x%x", privp->els_code);
6359     } else if (privp->fpkt->fcal_pkt_comp == sf_reportlun_callback) {
6360         if (sf_core && (sf_core & SF_CORE_REPORTLUN_TIMEOUT)) {
6361             sf_token = (int *) (uintptr_t) fpkt->fcal_social_request.\
6362                 sr_soc_hdr.sh_request_token;
6363             (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
6364             sf_core = 0;
6365         }
6366         timeout = SF_FCP_TIMEOUT;
6367         (void) sprintf(what, "REPORT LUNS");
6368     } else if (privp->fpkt->fcal_pkt_comp == sf_inq_callback) {
6369         if (sf_core && (sf_core & SF_CORE_INQUIRY_TIMEOUT)) {
6370             sf_token = (int *) (uintptr_t)
6371                 fpkt->fcal_social_request.\
6372                 sr_soc_hdr.sh_request_token;
6373             (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);
6374             sf_core = 0;
6375         }
6376         timeout = SF_FCP_TIMEOUT;
6377         (void) sprintf(what, "INQUIRY to LUN 0x%lx",
6378             (long)SCSA_LUN(target));
6379     } else {
6380         (void) sprintf(what, "UNKNOWN OPERATION");
6381     }

6383     if (dflag) {
6384         /* delayed retry */
6385         SF_DEBUG(2, (sf, CE_CONT,
6386             "!sf%d: %s to target %x delayed retry\n",
6387             ddi_get_instance(sf->sf_dip), what,
6388             sf_alpa_to_switch[privp->dest_nport_id]));
6389         privp->delayed_retry = FALSE;
6390         goto try_again;
6391     }

6393     sf_log(sf, CE_NOTE, "!%s to target 0x%x alpa 0x%x timed out\n",
6394         what, sf_alpa_to_switch[privp->dest_nport_id],

```

```

6395         privp->dest_nport_id);

6397         rval = soc_abort(sf->sf_sochandle, sf->sf_socp, sf->sf_sochandle
6398             ->fcal_portno, fpkt, 1);
6399         if (rval == FCAL_ABORTED || rval == FCAL_ABORT_FAILED) {
6400             SF_DEBUG(1, (sf, CE_NOTE, "!%s abort to al_pa %x succeeded\n",
6401                 what, privp->dest_nport_id));
6402         try_again:

6404             mutex_enter(&sf->sf_mutex);
6405             if (privp->prev != NULL) {
6406                 privp->prev->next = privp->next;
6407             }
6408             if (sf->sf_els_list == privp) {
6409                 sf->sf_els_list = privp->next;
6410             }
6411             if (privp->next != NULL) {
6412                 privp->next->prev = privp->prev;
6413             }
6414             privp->prev = privp->next = NULL;
6415             if (lip_cnt == sf->sf_lip_cnt) {
6416                 privp->timeout = sf_watchdog_time + timeout;
6417                 if ((++(privp->retries) < sf_els_retries) ||
6418                     (dflag && (privp->retries < SF_BSY_RETRIES))) {
6419                     mutex_exit(&sf->sf_mutex);
6420                     sf_log(sf, CE_NOTE,
6421                         "%s to target 0x%x retrying\n",
6422                         what,
6423                         sf_alpa_to_switch[privp->dest_nport_id]);
6424                     if (sf_els_transport(sf, privp) == 1) {
6425                         mutex_enter(&sf->sf_mutex);
6426                         return (sf->sf_els_list); /* success */
6427                     }
6428                     mutex_enter(&sf->sf_mutex);
6429                     fpkt = NULL;
6430                 }
6431             }
6432             if ((lip_cnt == sf->sf_lip_cnt) &&
6433                 (els_code != LA_ELS_LOGO)) {
6434                 if (target != NULL) {
6435                     sf_offline_target(sf, target);
6436                 }
6437                 if (sf->sf_lip_cnt == lip_cnt) {
6438                     sf->sf_device_count--;
6439                     ASSERT(sf->sf_device_count >= 0);
6440                     if (sf->sf_device_count == 0) {
6441                         sf_finish_init(sf,
6442                             sf->sf_lip_cnt);
6443                     }
6444                 }
6445             }
6446             privp = sf->sf_els_list;
6447             mutex_exit(&sf->sf_mutex);
6448             if (fpkt != NULL) {
6449                 sf_els_free(fpkt);
6450             }
6451         } else {
6452             mutex_exit(&sf->sf_mutex);
6453             sf_els_free(privp->fpkt);
6454             privp = NULL;
6455         }
6456     } else {
6457         if (sf_core && (sf_core & SF_CORE_ELS_FAILED)) {
6458             sf_token = (int *) (uintptr_t)
6459                 fpkt->fcal_social_request.\
6460                 sr_soc_hdr.sh_request_token;
6461             (void) soc_take_core(sf->sf_sochandle, sf->sf_socp);

```

```

6461         sf_core = 0;
6462     }
6463     sf_log(sf, CE_NOTE, "%s abort to target 0x%x failed. "
6464           "status=0x%x, forcing LIP\n", what,
6465           sf_alpa_to_switch[privp->dest_nport_id], rval);
6466     privp = NULL;
6467     if (sf->sf_lip_cnt == lip_cnt) {
6468         sf_force_lip(sf);
6469     }
6470 }

6472     mutex_enter(&sf->sf_mutex);
6473     return (privp);
6474 }

6477 /*
6478  * called by timeout when a reset times out
6479  */
6480 /*ARGSUSED*/
6481 static void
6482 sf_check_reset_delay(void *arg)
6483 {
6484     struct sf *sf;
6485     struct sf_target *target;
6486     struct sf_reset_list *rp, *tp;
6487     uint_t lip_cnt, reset_timeout_flag = FALSE;
6488     clock_t lb;

6490     lb = ddi_get_lbolt();

6492     mutex_enter(&sf_global_mutex);

6494     sf_reset_timeout_id = 0;

6496     for (sf = sf_head; sf != NULL; sf = sf->sf_next) {

6498         mutex_exit(&sf_global_mutex);
6499         mutex_enter(&sf->sf_mutex);

6501         /* is this type cast needed? */
6502         tp = (struct sf_reset_list *)&sf->sf_reset_list;

6504         rp = sf->sf_reset_list;
6505         while (rp != NULL) {
6506             if (((rp->timeout - lb) < 0) &&
6507                 (rp->lip_cnt == sf->sf_lip_cnt)) {
6508                 tp->next = rp->next;
6509                 mutex_exit(&sf->sf_mutex);
6510                 target = rp->target;
6511                 lip_cnt = rp->lip_cnt;
6512                 kmem_free(rp, sizeof (struct sf_reset_list));
6513                 /* abort all cmds for this target */
6514                 while (target) {
6515                     sf_abort_all(sf, target, FALSE,
6516                                 lip_cnt, TRUE);
6517                     mutex_enter(&target->sft_mutex);
6518                     if (lip_cnt == sf->sf_lip_cnt) {
6519                         target->sft_state &=
6520                             ~SF_TARGET_BUSY;
6521                     }
6522                     mutex_exit(&target->sft_mutex);
6523                     target = target->sft_next_lun;
6524                 }
6525                 mutex_enter(&sf->sf_mutex);
6526                 tp = (struct sf_reset_list *)

```

```

6527         &sf->sf_reset_list;
6528         rp = sf->sf_reset_list;
6529         lb = ddi_get_lbolt();
6530     } else if (rp->lip_cnt != sf->sf_lip_cnt) {
6531         tp->next = rp->next;
6532         kmem_free(rp, sizeof (struct sf_reset_list));
6533         rp = tp->next;
6534     } else {
6535         reset_timeout_flag = TRUE;
6536         tp = rp;
6537         rp = rp->next;
6538     }
6539 }
6540     mutex_exit(&sf->sf_mutex);
6541     mutex_enter(&sf_global_mutex);
6542 }

6544     if (reset_timeout_flag && (sf_reset_timeout_id == 0)) {
6545         sf_reset_timeout_id = timeout(sf_check_reset_delay,
6546                                     NULL, drv_usectohz(SF_TARGET_RESET_DELAY));
6547     }

6549     mutex_exit(&sf_global_mutex);
6550 }

6553 /*
6554  * called to "reset the bus", i.e. force loop initialization (and address
6555  * re-negotiation)
6556  */
6557 static void
6558 sf_force_lip(struct sf *sf)
6559 {
6560     int i;
6561     struct sf_target *target;

6564     /* disable restart of lip if we're suspended */
6565     mutex_enter(&sf->sf_mutex);
6566     if (sf->sf_state & SF_STATE_SUSPENDED) {
6567         mutex_exit(&sf->sf_mutex);
6568         SF_DEBUG(1, (sf, CE_CONT,
6569                    "sf_force_lip, sf%d: lip restart disabled "
6570                    "due to DDI_SUSPEND\n",
6571                    ddi_get_instance(sf->sf_dip)));
6572         return;
6573     }

6575     sf_log(sf, CE_NOTE, "Forcing lip\n");

6577     for (i = 0; i < sf_max_targets; i++) {
6578         target = sf->sf_targets[i];
6579         while (target != NULL) {
6580             mutex_enter(&target->sft_mutex);
6581             if (!(target->sft_state & SF_TARGET_OFFLINE))
6582                 target->sft_state |= SF_TARGET_BUSY;
6583             mutex_exit(&target->sft_mutex);
6584             target = target->sft_next_lun;
6585         }
6586     }

6588     sf->sf_lip_cnt++;
6589     sf->sf_timer = sf_watchdog_time + SF_OFFLINE_TIMEOUT;
6590     sf->sf_state = SF_STATE_OFFLINE;
6591     mutex_exit(&sf->sf_mutex);
6592     sf->sf_stats.lip_count++; /* no mutex for this? */

```

```

6594 #ifndef DEBUG
6595     /* are we allowing LIPs ?? */
6596     if (sf_lip_flag != 0) {
6597 #endif
6598         /* call the transport to force loop initialization */
6599         if (((i = soc_force_lip(sf->sf_sochandle, sf->sf_socp,
6600             sf->sf_sochandle->fcal_portno, 1,
6601             FCAL_FORCE_LIP)) != FCAL_SUCCESS) &&
6602             (i != FCAL_TIMEOUT)) {
6603             /* force LIP failed */
6604             if (sf_core && (sf_core & SF_CORE_LIP_FAILED)) {
6605                 (void) soc_take_core(sf->sf_sochandle,
6606                     sf->sf_socp);
6607                 sf_core = 0;
6608             }
6609 #ifndef DEBUG
6610             /* are we allowing reset after LIP failed ?? */
6611             if (sf_reset_flag != 0) {
6612 #endif
6613                 /* restart social after resetting it */
6614                 sf_log(sf, CE_NOTE,
6615                     "!Force lip failed Status code 0x%x."
6616                     " Reseting\n", i);
6617                 /* call transport to force a reset */
6618                 soc_force_reset(sf->sf_sochandle, sf->sf_socp,
6619                     sf->sf_sochandle->fcal_portno, 1);
6620 #ifndef DEBUG
6621             }
6622 #endif
6623         }
6624 #ifndef DEBUG
6625     }
6626 #endif
6627 }

6630 /*
6631  * called by the transport when an unsolicited ELS is received
6632  */
6633 static void
6634 sf_unsol_els_callback(void *arg, soc_response_t *srp, caddr_t payload)
6635 {
6636     struct sf *sf = (struct sf *)arg;
6637     els_payload_t *els = (els_payload_t *)payload;
6638     struct la_els_rjt *rsp;
6639     int i, tgt_id;
6640     uchar_t dest_id;
6641     struct fcal_packet *fpkt;
6642     fc_frame_header_t *hp;
6643     struct sf_els_hdr *privp;

6646     if ((els == NULL) || ((i = srp->sr_soc_hdr.sh_byte_cnt) == 0)) {
6647         return;
6648     }

6650     if (i > SOC_CQE_PAYLOAD) {
6651         i = SOC_CQE_PAYLOAD;
6652     }

6654     dest_id = (uchar_t)srp->sr_fc_frame_hdr.s_id;
6655     tgt_id = sf_alpa_to_switch[dest_id];

6657     switch (els->els_cmd.c.ls_command) {

```

```

6659     case LA_ELS_LOGO:
6660         /*
6661          * logout received -- log the fact
6662          */
6663         sf->sf_stats.tstats[tgt_id].logouts_recvd++;
6664         sf_log(sf, CE_NOTE, "!LOGO recvd from target %x, %s\n",
6665             tgt_id,
6666             sf_lip_on_plogo ? "Forcing LIP..." : "");
6667         if (sf_lip_on_plogo) {
6668             sf_force_lip(sf);
6669         }
6670         break;

6672     default: /* includes LA_ELS_PLOGI */
6673         /*
6674          * something besides a logout received -- we don't handle
6675          * this so send back a reject saying its unsupported
6676          */
6677         sf_log(sf, CE_NOTE, "!ELS 0x%x recvd from target 0x%x\n",
6678             els->els_cmd.c.ls_command, tgt_id);

6682         /* allocate room for a response */
6683         if (sf_els_alloc(sf, dest_id, sizeof (struct sf_els_hdr),
6684             sizeof (struct la_els_rjt), sizeof (union sf_els_rsp),
6685             (caddr_t *)&privp, (caddr_t *)&rsp) == NULL) {
6686             break;
6687         }

6689         fpkt = privp->fpkt;

6691         /* fill in pkt header */
6692         hp = &fpkt->fcal_socal_request.sr_fc_frame_hdr;
6693         hp->r_ctl = R_CTL_ELS_RSP;
6694         hp->f_ctl = F_CTL_LAST_SEQ | F_CTL_XCHG_CONTEXT;
6695         hp->ox_id = srp->sr_fc_frame_hdr.ox_id;
6696         hp->rx_id = srp->sr_fc_frame_hdr.rx_id;
6697         fpkt->fcal_socal_request.sr_cqhdr.cq_hdr_type =
6698             CQ_TYPE_OUTBOUND;

6700         fpkt->fcal_socal_request.sr_soc_hdr.sh_seg_cnt = 1;

6702         /* fill in response */
6703         rsp->ls_code = LA_ELS_RJT; /* reject this ELS */
6704         rsp->mbz[0] = 0;
6705         rsp->mbz[1] = 0;
6706         rsp->mbz[2] = 0;
6707         ((struct la_els_logi *)privp->rsp)->ls_code = LA_ELS_ACC;
6708         *((int *)&rsp->reserved) = 0;
6709         rsp->reason_code = RJT_UNSUPPOTED;
6710         privp->retries = sf_els_retries;
6711         privp->els_code = LA_ELS_RJT;
6712         privp->timeout = (unsigned)0xffffffff;
6713         (void) sf_els_transport(sf, privp);
6714         break;
6715     }
6716 }

6719 /*
6720  * Error logging, printing, and debug print routines
6721  */

6723 /*PRINTFLIKE3*/
6724 static void

```

```

6725 sf_log(struct sf *sf, int level, const char *fmt, ...)
6726 {
6727     char buf[256];
6728     dev_info_t *dip;
6729     va_list ap;

6731     if (sf != NULL) {
6732         dip = sf->sf_dip;
6733     } else {
6734         dip = NULL;
6735     }

6737     va_start(ap, fmt);
6738     (void) vsprintf(buf, fmt, ap);
6739     va_end(ap);
6740     scsi_log(dip, "sf", level, buf);
6741 }

6744 /*
6745  * called to get some sf kstats -- return 0 on success else return errno
6746  */
6747 static int
6748 sf_kstat_update(kstat_t *ksp, int rw)
6749 {
6750     struct sf *sf;

6752     if (rw == KSTAT_WRITE) {
6753         /* can't write */
6754         return (EACCES);
6755     }

6757     sf = ksp->ks_private;
6758     sf->sf_stats.ncmds = sf->sf_ncmds;
6759     sf->sf_stats.throttle_limit = sf->sf_throttle;
6760     sf->sf_stats.cr_pool_size = sf->sf_cr_pool_cnt;

6762     return (0); /* success */
6763 }

6766 /*
6767  * Unix Entry Points
6768  */

6770 /*
6771  * driver entry point for opens on control device
6772  */
6773 /* ARGSUSED */
6774 static int
6775 sf_open(dev_t *dev_p, int flag, int otyp, cred_t *cred_p)
6776 {
6777     dev_t dev = *dev_p;
6778     struct sf *sf;

6781     /* just ensure soft state exists for this device */
6782     sf = ddi_get_soft_state(sf_state, SF_MINOR2INST(getminor(dev)));
6783     if (sf == NULL) {
6784         return (ENXIO);
6785     }

6787     ++(sf->sf_check_n_close);

6789     return (0);
6790 }

```

```

6793 /*
6794  * driver entry point for last close on control device
6795  */
6796 /* ARGSUSED */
6797 static int
6798 sf_close(dev_t dev, int flag, int otyp, cred_t *cred_p)
6799 {
6800     struct sf *sf;

6802     sf = ddi_get_soft_state(sf_state, SF_MINOR2INST(getminor(dev)));
6803     if (sf == NULL) {
6804         return (ENXIO);
6805     }

6807     if (!sf->sf_check_n_close) { /* if this flag is zero */
6808         cmn_err(CE_WARN, "sf%d: trying to close unopened instance",
6809             SF_MINOR2INST(getminor(dev)));
6810         return (ENODEV);
6811     } else {
6812         --(sf->sf_check_n_close);
6813     }
6814     return (0);
6815 }

6818 /*
6819  * driver entry point for sf ioctl commands
6820  */
6821 /* ARGSUSED */
6822 static int
6823 sf_ioctl(dev_t dev,
6824     int cmd, intp_t arg, int mode, cred_t *cred_p, int *rval_p)
6825 {
6826     struct sf *sf;
6827     struct sf_target *target;
6828     uchar_t al_pa;
6829     struct sf_al_map map;
6830     int cnt, i;
6831     int     retval; /* return value */
6832     struct devctl_iocdata *dcp;
6833     dev_info_t *cdip;
6834     struct scsi_address ap;
6835     scsi_hba_tran_t *tran;

6838     sf = ddi_get_soft_state(sf_state, SF_MINOR2INST(getminor(dev)));
6839     if (sf == NULL) {
6840         return (ENXIO);
6841     }

6843     /* handle all ioctls */
6844     switch (cmd) {

6846     /*
6847      * We can use the generic implementation for these ioctls
6848      */
6849     case DEVCTL_DEVICE_GETSTATE:
6850     case DEVCTL_DEVICE_ONLINE:
6851     case DEVCTL_DEVICE_OFFLINE:
6852     case DEVCTL_BUS_GETSTATE:
6853         return (ndi_devctl_ioctl(sf->sf_dip, cmd, arg, mode, 0));

6855     /*
6856      * return FC map

```

```

6857  */
6858  case SFIOCGMAP:
6859      if ((sf->sf_lilp_map->lilp_magic != FCAL_LILP_MAGIC &&
6860          sf->sf_lilp_map->lilp_magic != FCAL_BADLILP_MAGIC) ||
6861          sf->sf_state != SF_STATE_ONLINE) {
6862          retval = ENOENT;
6863          goto dun;
6864      }
6865      mutex_enter(&sf->sf_mutex);
6866      if (sf->sf_lilp_map->lilp_magic == FCAL_BADLILP_MAGIC) {
6867          int i, j = 0;
6868
6869          /* Need to generate a fake lilp map */
6870          for (i = 0; i < sf_max_targets; i++) {
6871              if (sf->sf_targets[i])
6872                  sf->sf_lilp_map->lilp_alpalist[j++] =
6873                      sf->sf_targets[i]->
6874                          sft_hard_address;
6875          }
6876          sf->sf_lilp_map->lilp_length = (uchar_t)j;
6877      }
6878      cnt = sf->sf_lilp_map->lilp_length;
6879      map.sf_count = (short)cnt;
6880      bcopy((caddr_t)&sf->sf_sochandle->fcal_n_wnn,
6881          (caddr_t)&map.sf_hba_addr.sf_node_wnn,
6882          sizeof (la_wnn_t));
6883      bcopy((caddr_t)&sf->sf_sochandle->fcal_p_wnn,
6884          (caddr_t)&map.sf_hba_addr.sf_port_wnn,
6885          sizeof (la_wnn_t));
6886      map.sf_hba_addr.sf_al_pa = sf->sf_al_pa;
6887      map.sf_hba_addr.sf_hard_address = 0;
6888      map.sf_hba_addr.sf_inq_dtype = DTYPE_UNKNOWN;
6889      for (i = 0; i < cnt; i++) {
6890          al_pa = sf->sf_lilp_map->lilp_alpalist[i];
6891          map.sf_addr_pair[i].sf_al_pa = al_pa;
6892          if (al_pa == sf->sf_al_pa) {
6893              (void) bcopy((caddr_t)&sf->sf_sochandle
6894                  ->fcal_n_wnn, (caddr_t)&map.
6895                      sf_addr_pair[i].sf_node_wnn,
6896                      sizeof (la_wnn_t));
6897              (void) bcopy((caddr_t)&sf->sf_sochandle
6898                  ->fcal_p_wnn, (caddr_t)&map.
6899                      sf_addr_pair[i].sf_port_wnn,
6900                      sizeof (la_wnn_t));
6901              map.sf_addr_pair[i].sf_hard_address =
6902                  al_pa;
6903              map.sf_addr_pair[i].sf_inq_dtype =
6904                  DTYPE_PROCESSOR;
6905              continue;
6906          }
6907          target = sf->sf_targets[sf_alpa_to_switch[
6908              al_pa]];
6909          if (target != NULL) {
6910              mutex_enter(&target->sft_mutex);
6911              if (!(target->sft_state &
6912                  (SF_TARGET_OFFLINE |
6913                  SF_TARGET_BUSY))) {
6914                  bcopy((caddr_t)&target->
6915                      sft_node_wnn,
6916                      (caddr_t)&map.sf_addr_pair
6917                          [i].sf_node_wnn,
6918                      sizeof (la_wnn_t));
6919                  bcopy((caddr_t)&target->
6920                      sft_port_wnn,
6921                      (caddr_t)&map.sf_addr_pair
6922                          [i].sf_port_wnn,

```

```

6923          sizeof (la_wnn_t));
6924          map.sf_addr_pair[i].
6925              sf_hard_address
6926              = target->sft_hard_address;
6927          map.sf_addr_pair[i].
6928              sf_inq_dtype
6929              = target->sft_device_type;
6930          mutex_exit(&target->sft_mutex);
6931          continue;
6932      }
6933      mutex_exit(&target->sft_mutex);
6934  }
6935  bzero((caddr_t)&map.sf_addr_pair[i].
6936      sf_node_wnn, sizeof (la_wnn_t));
6937  bzero((caddr_t)&map.sf_addr_pair[i].
6938      sf_port_wnn, sizeof (la_wnn_t));
6939  map.sf_addr_pair[i].sf_inq_dtype =
6940      DTYPE_UNKNOWN;
6941  }
6942  mutex_exit(&sf->sf_mutex);
6943  if (ddi_copyout((caddr_t)&map, (caddr_t)arg,
6944      sizeof (struct sf_al_map), mode) != 0) {
6945      retval = EFAULT;
6946      goto dun;
6947  }
6948  break;
6949
6950  /*
6951  * handle device control ioctls
6952  */
6953  case DEVCTL_DEVICE_RESET:
6954      if (ndi_dc_allochdl((void *)arg, &dc) != NDI_SUCCESS) {
6955          retval = EFAULT;
6956          goto dun;
6957      }
6958      if ((ndi_dc_getname(dc) == NULL) ||
6959          (ndi_dc_getaddr(dc) == NULL)) {
6960          ndi_dc_freehdl(dc);
6961          retval = EINVAL;
6962          goto dun;
6963      }
6964      cdip = ndi_devi_find(sf->sf_dip,
6965          ndi_dc_getname(dc), ndi_dc_getaddr(dc));
6966      ndi_dc_freehdl(dc);
6967
6968      if (cdip == NULL) {
6969          retval = ENXIO;
6970          goto dun;
6971      }
6972
6973      if ((target = sf_get_target_from_dip(sf, cdip)) == NULL) {
6974          retval = ENXIO;
6975          goto dun;
6976      }
6977      mutex_enter(&target->sft_mutex);
6978      if (!(target->sft_state & SF_TARGET_INIT_DONE)) {
6979          mutex_exit(&target->sft_mutex);
6980          retval = ENXIO;
6981          goto dun;
6982      }
6983
6984      /* This is ugly */
6985      tran = kmem_zalloc(scsi_hba_tran_size(), KM_SLEEP);
6986      bcopy(target->sft_tran, tran, scsi_hba_tran_size());
6987      mutex_exit(&target->sft_mutex);
6988      ap.a_hba_tran = tran;

```



```

6989         ap.a_target = sf_alpa_to_switch[target->sft_al_pa];
6990         if (sf_reset(&ap, RESET_TARGET) == FALSE) {
6991             retval = EIO;
6992         } else {
6993             retval = 0;
6994         }
6995         kmem_free(tran, scsi_hba_tran_size());
6996         goto dun;

6998     case DEVCTL_BUS_QUIESCE:
6999     case DEVCTL_BUS_UNQUIESCE:
7000         retval = ENOTSUP;
7001         goto dun;

7003     case DEVCTL_BUS_RESET:
7004     case DEVCTL_BUS_RESETELL:
7005         sf_force_lip(sf);
7006         break;

7008     default:
7009         retval = ENOTTY;
7010         goto dun;
7011     }

7013     retval = 0;                /* success */

7015 dun:
7016     return (retval);
7017 }

7020 /*
7021  * get the target given a DIP
7022  */
7023 static struct sf_target *
7024 sf_get_target_from_dip(struct sf *sf, dev_info_t *dip)
7025 {
7026     int i;
7027     struct sf_target *target;

7030     /* scan each hash queue for the DIP in question */
7031     for (i = 0; i < SF_NUM_HASH_QUEUES; i++) {
7032         target = sf->sf_wnn_lists[i];
7033         while (target != NULL) {
7034             if (target->sft_dip == dip) {
7035                 return (target); /* success: target found */
7036             }
7037             target = target->sft_next;
7038         }
7039     }
7040     return (NULL);            /* failure: target not found */
7041 }

7044 /*
7045  * called by the transport to get an event cookie
7046  */
7047 static int
7048 sf_bus_get_eventcookie(dev_info_t *dip, dev_info_t *rdip, char *name,
7049     ddi_eventcookie_t *event_cookiep)
7050 {
7051     struct sf *sf;

7053     sf = ddi_get_soft_state(sf_state, ddi_get_instance(dip));
7054     if (sf == NULL) {

```

```

7055         /* can't find instance for this device */
7056         return (DDI_FAILURE);
7057     }

7059     return (ndi_event_retrieve_cookie(sf->sf_event_hdl, rdip, name,
7060         event_cookiep, NDI_EVENT_NOPASS));
7062 }

7065 /*
7066  * called by the transport to add an event callback
7067  */
7068 static int
7069 sf_bus_add_eventcall(dev_info_t *dip, dev_info_t *rdip,
7070     ddi_eventcookie_t eventid, void (*callback)(dev_info_t *dip,
7071     ddi_eventcookie_t event, void *arg, void *impl_data), void *arg,
7072     ddi_callback_id_t *cb_id)
7073 {
7074     struct sf *sf;

7076     sf = ddi_get_soft_state(sf_state, ddi_get_instance(dip));
7077     if (sf == NULL) {
7078         /* can't find instance for this device */
7079         return (DDI_FAILURE);
7080     }

7082     return (ndi_event_add_callback(sf->sf_event_hdl, rdip,
7083         eventid, callback, arg, NDI_SLEEP, cb_id));
7085 }

7088 /*
7089  * called by the transport to remove an event callback
7090  */
7091 static int
7092 sf_bus_remove_eventcall(dev_info_t *devi, ddi_callback_id_t cb_id)
7093 {
7094     struct sf *sf;

7096     sf = ddi_get_soft_state(sf_state, ddi_get_instance(devi));
7097     if (sf == NULL) {
7098         /* can't find instance for this device */
7099         return (DDI_FAILURE);
7100     }

7102     return (ndi_event_remove_callback(sf->sf_event_hdl, cb_id));
7103 }

7106 /*
7107  * called by the transport to post an event
7108  */
7109 static int
7110 sf_bus_post_event(dev_info_t *dip, dev_info_t *rdip,
7111     ddi_eventcookie_t eventid, void *impldata)
7112 {
7113     ddi_eventcookie_t remove_cookie, cookie;

7115     /* is this a remove event ?? */
7116     struct sf *sf = ddi_get_soft_state(sf_state, ddi_get_instance(dip));
7117     remove_cookie = ndi_event_tag_to_cookie(sf->sf_event_hdl,
7118         SF_EVENT_TAG_REMOVE);
7118     if (remove_cookie == eventid) {

```

```

7121     struct sf_target *target;
7122
7123     /* handle remove event */
7124
7125     if (sf == NULL) {
7126         /* no sf instance for this device */
7127         return (NDI_FAILURE);
7128     }
7129
7130     /* get the target for this event */
7131     if ((target = sf_get_target_from_dip(sf, rdip)) != NULL) {
7132         /*
7133          * clear device info for this target and mark as
7134          * not done
7135          */
7136         mutex_enter(&target->sft_mutex);
7137         target->sft_dip = NULL;
7138         target->sft_state &= ~SF_TARGET_INIT_DONE;
7139         mutex_exit(&target->sft_mutex);
7140         return (NDI_SUCCESS); /* event handled */
7141     }
7142
7143     /* no target for this event */
7144     return (NDI_FAILURE);
7145 }
7146
7147 /* an insertion event */
7148 if (ndi_busop_get_eventcookie(dip, rdip, FCAL_INSERT_EVENT, &cookie)
7149     != NDI_SUCCESS) {
7150     return (NDI_FAILURE);
7151 }
7152
7153 return (ndi_post_event(dip, rdip, cookie, impldata));
7154 }
7155
7157 /*
7158  * the sf hotplug daemon, one thread per sf instance
7159  */
7160 static void
7161 sf_hp_daemon(void *arg)
7162 {
7163     struct sf *sf = (struct sf *)arg;
7164     struct sf_hp_elem *elem;
7165     struct sf_target *target;
7166     int tgt_id;
7167     callb_cpr_t cprinfo;
7168
7169     CALLB_CPR_INIT(&cprinfo, &sf->sf_hp_daemon_mutex,
7170                  callb_generic_cpr, "sf_hp_daemon");
7171
7172     mutex_enter(&sf->sf_hp_daemon_mutex);
7173
7174     do {
7175         while (sf->sf_hp_elem_head != NULL) {
7176
7177             /* save ptr to head of list */
7178             elem = sf->sf_hp_elem_head;
7179
7180             /* take element off of list */
7181             if (sf->sf_hp_elem_head == sf->sf_hp_elem_tail) {
7182                 /* element only one in list -- list now empty */
7183                 sf->sf_hp_elem_head = NULL;
7184                 sf->sf_hp_elem_tail = NULL;
7185             } else {
7186                 /* remove element from head of list */

```

```

7187         sf->sf_hp_elem_head = sf->sf_hp_elem_head->next;
7188     }
7189
7190     mutex_exit(&sf->sf_hp_daemon_mutex);
7191
7192     switch (elem->what) {
7193     case SF_ONLINE:
7194         /* online this target */
7195         target = elem->target;
7196         (void) ndi_devi_online(elem->dip, 0);
7197         (void) ndi_event_retrieve_cookie(
7198             sf->sf_event_hdl,
7199             target->sft_dip, FCAL_INSERT_EVENT,
7200             &sf_insert_eid, NDI_EVENT_NOPASS);
7201         (void) ndi_event_run_callbacks(sf->sf_event_hdl,
7202             target->sft_dip, sf_insert_eid, NULL);
7203         break;
7204     case SF_OFFLINE:
7205         /* offline this target */
7206         target = elem->target;
7207         tgt_id = sf_alpa_to_switch[target->sft_al_pa];
7208         /* don't do NDI_DEVI_REMOVE for now */
7209         if (ndi_devi_offline(elem->dip, 0) !=
7210             NDI_SUCCESS) {
7211             SF_DEBUG(1, (sf, CE_WARN, "target %x, "
7212                 "device offline failed", tgt_id));
7213         } else {
7214             SF_DEBUG(1, (sf, CE_NOTE, "target %x, "
7215                 "device offline succeeded\n",
7216                 tgt_id));
7217         }
7218         break;
7219     }
7220     kmem_free(elem, sizeof (struct sf_hp_elem));
7221     mutex_enter(&sf->sf_hp_daemon_mutex);
7222 }
7223
7224 /* if exit is not already signaled */
7225 if (sf->sf_hp_exit == 0) {
7226     /* wait to be signaled by work or exit */
7227     CALLB_CPR_SAFE_BEGIN(&cprinfo);
7228     cv_wait(&sf->sf_hp_daemon_cv, &sf->sf_hp_daemon_mutex);
7229     CALLB_CPR_SAFE_END(&cprinfo, &sf->sf_hp_daemon_mutex);
7230 }
7231 } while (sf->sf_hp_exit == 0);
7232
7233 /* sf_hp_daemon_mutex is dropped by CALLB_CPR_EXIT */
7234 CALLB_CPR_EXIT(&cprinfo);
7235 thread_exit(); /* no more hotplug thread */
7236 /* NOTREACHED */
7237 }

```

```
*****
```

```
99351 Thu Feb 25 15:39:46 2016
```

```
new/usr/src/uts/sun4u/io/rmclomv.c
```

```
2976 remove useless offsetof() macros
```

```
*****
```

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */
```

```
28 #include <sys/types.h>
29 #include <sys/stat.h>
30 #include <sys/conf.h>
31 #include <sys/modctl.h>
32 #include <sys/callb.h>
33 #include <sys/strlog.h>
34 #include <sys/cyclic.h>
35 #include <sys/rmc_comm_dp.h>
36 #include <sys/rmc_comm_dp_boot.h>
37 #include <sys/rmc_comm_drvintf.h>
38 #include <sys/rmc_comm.h>
39 #include <sys/machsystem.h>
40 #include <sys/sysevent.h>
41 #include <sys/sysevent/dr.h>
42 #include <sys/sysevent/env.h>
43 #include <sys/sysevent/eventdefs.h>
44 #include <sys/file.h>
45 #include <sys/disp.h>
46 #include <sys/reboot.h>
47 #include <sys/envmon.h>
48 #include <sys/rmclomv_impl.h>
49 #include <sys/cpu_sgnblk_defs.h>
50 #include <sys/utsname.h>
51 #include <sys/systeminfo.h>
52 #include <sys/ddi.h>
53 #include <sys/time.h>
54 #include <sys/promif.h>
55 #include <sys/sysmacros.h>
56 #endif /* ! codereview */

58 #define offsetof(s, m) ((size_t)((s *)0->m))
59 #define RMCRESBUFLEN 1024
60 #define DATE_TIME_MSG_SIZE 78
61 #define RMCLOMV_WATCHDOG_MODE "rmclomv-watchdog-mode"
```

```
61 #define DELAY_TIME 500000 /* 5 seconds, in microseconds */
62 #define CPU_SIGNATURE_DELAY_TIME 5000000 /* 5 secs, in microsecs */

64 extern void pmugpio_watchdog_pat();

66 extern int watchdog_activated;
67 static int last_watchdog_msg = 1;
68 extern int watchdog_enable;
69 extern int boothowto;

71 int rmclomv_watchdog_mode;

73 /*
74  * functions local to this driver.
75  */
76 static int rmclomv_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
77 void **resultp);
78 static int rmclomv_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
79 static int rmclomv_detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
80 static uint_t rmclomv_break_intr(caddr_t arg);
81 static int rmclomv_add_intr_handlers(void);
82 static int rmclomv_remove_intr_handlers(void);
83 static uint_t rmclomv_event_data_handler(char *);
84 static void rmclomv_dr_data_handler(const char *, int);
85 static int rmclomv_open(dev_t *dev_p, int flag, int otyp, cred_t *cred_p);
86 static int rmclomv_close(dev_t dev, int flag, int otyp, cred_t *cred_p);
87 static int rmclomv_ioctl(dev_t dev, int cmd, intp_t arg, int mode,
88 cred_t *cred_p, int *rval_p);
89 static void rmclomv_checkrmc_start(void);
90 static void rmclomv_checkrmc_destroy(void);
91 static void rmclomv_checkrmc_wakeup(void *);
92 static void rmclomv_refresh_start(void);
93 static void rmclomv_refresh_destroy(void);
94 static void rmclomv_refresh_wakeup(void);
95 static void rmclomv_reset_cache(rmclomv_cache_section_t *new_chain,
96 rmclomv_cache_section_t *new_subchain, dp_get_sysinfo_r_t *sysinfo);
97 static rmclomv_cache_section_t *rmclomv_find_section(
98 rmclomv_cache_section_t *start, uint16_t sensor);
99 static rmclomv_cache_section_t *create_cache_section(int sensor_type, int num);
100 static int get_sensor_by_name(const rmclomv_cache_section_t *section,
101 const char *name, int *index);
102 static int validate_section_entry(rmclomv_cache_section_t *section,
103 int index);
104 static int add_names_to_section(rmclomv_cache_section_t *section);
105 static void free_section(rmclomv_cache_section_t *section);
106 static void add_section(rmclomv_cache_section_t **head,
107 rmclomv_cache_section_t *section);
108 static int rmclomv_do_cmd(int req_cmd, int resp_cmd, int resp_len,
109 intp_t arg_req, intp_t arg_res);
110 static void refresh_name_cache(int force_fail);
111 static void set_val_unav(envmon_sensor_t *sensor);
112 static void set_fan_unav(envmon_fan_t *fan);
113 static int do_psu_cmd(intp_t arg, int mode, envmon_indicator_t *env_ind,
114 dp_get_psu_status_t *rmc_psu, dp_get_psu_status_r_t *rmc_psu_r,
115 int detector_type);
116 static uint_t rmc_set_watchdog_timer(uint_t timeoutval);
117 static uint_t rmc_clear_watchdog_timer(void);
118 static void send_watchdog_msg(int msg);
119 static void plat_timesync(void *arg);

121 static kmutex_t timesync_lock;
122 static clock_t timesync_interval = 0;
123 static timeout_id_t timesync_tid = 0;

125 /*
126  * Driver entry points
```

```
127 */
128 static struct cb_ops rmclomv_cb_ops = {
129     rmclomv_open,    /* open */
130     rmclomv_close,  /* close */
131     nodev,          /* strategy() */
132     nodev,          /* print() */
133     nodev,          /* dump() */
134     nodev,          /* read() */
135     nodev,          /* write() */
136     rmclomv_ioctl,  /* ioctl() */
137     nodev,          /* devmap() */
138     nodev,          /* mmap() */
139     ddi_segmap,     /* segmap() */
140     nochpoll,       /* poll() */
141     ddi_prop_op,    /* prop_op() */
142     NULL,           /* cb_str */
143     D_NEW | D_MP    /* cb_flag */
144 };
unchanged_portion_omitted
```