```
**********************************************************
  104952 Tue Sep 19 12:56:21 2017
new/usr/src/uts/common/io/nvme/nvme.c
don't block in nvme_bd_cmd
8629 nvme: rework command abortion
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
Reviewed by: Jason King <jason.king@joyent.com>
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2016 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2016 Tegile Systems, Inc. All rights reserved.
 * Copyright (c) 2016 The MathWorks, Inc.  All rights reserved.
 * Copyright 2017 Joyent, Inc.
 */

/*
 * blkdev driver for NVMe compliant storage devices
 *
 * This driver was written to conform to version 1.2.1 of the NVMe
 * specification.  It may work with newer versions, but that is completely
 * untested and disabled by default.
 *
 * The driver has only been tested on x86 systems and will not work on big-
 * endian systems without changes to the code accessing registers and data
 * structures used by the hardware.
 *
 *
 * Interrupt Usage:
 *
 * The driver will use a single interrupt while configuring the device as the
 * specification requires, but contrary to the specification it will try to use
 * a single-message MSI(-X) or FIXED interrupt. Later in the attach process it
 * will switch to multiple-message MSI(-X) if supported. The driver wants to
 * have one interrupt vector per CPU, but it will work correctly if less are
 * available. Interrupts can be shared by queues, the interrupt handler will
 * iterate through the I/O queue array by steps of n_intr_cnt. Usually only
 * the admin queue will share an interrupt with one I/O queue. The interrupt
 * handler will retrieve completed commands from all queues sharing an interrupt
 * vector and will post them to a taskq for completion processing.
 *
 *
 * Command Processing:
 *
 * NVMe devices can have up to 65535 I/O queue pairs, with each queue holding up
 * to 65536 I/O commands. The driver will configure one I/O queue pair per
 * available interrupt vector, with the queue length usually much smaller than
 * the maximum of 65536. If the hardware doesn't provide enough queues, fewer
 * interrupt vectors will be used.
 *
 * Additionally the hardware provides a single special admin queue pair that can
 * hold up to 4096 admin commands.
 *
 * From the hardware perspective both queues of a queue pair are independent,
 * but they share some driver state: the command array (holding pointers to
```



```
 * commands currently being processed by the hardware) and the active command
 * counter. Access to a queue pair and the shared state is protected by
 * nq_mutex.
 * counter. Access to the submission side of a queue pair and the shared state
 * is protected by nq_mutex. The completion side of a queue pair does not need
 * that protection apart from its access to the shared state; it is called only
 * in the interrupt handler which does not run concurrently for the same
 * interrupt vector.
 *
 * When a command is submitted to a queue pair the active command counter is
 * incremented and a pointer to the command is stored in the command array. The
 * array index is used as command identifier (CID) in the submission queue
 * entry. Some commands may take a very long time to complete, and if the queue
 * wraps around in that time a submission may find the next array slot to still
 * be used by a long-running command. In this case the array is sequentially
 * searched for the next free slot. The length of the command array is the same
 * as the configured queue length. Queue overrun is prevented by the semaphore,
 * so a command submission may block if the queue is full.
 *
 *
 * Polled I/O Support:
 *
 * For kernel core dump support the driver can do polled I/O. As interrupts are
 * turned off while dumping the driver will just submit a command in the regular
 * way, and then repeatedly attempt a command retrieval until it gets the
 * command back.
 *
 *
 * Namespace Support:
 *
 * NVMe devices can have multiple namespaces, each being a independent data
 * store. The driver supports multiple namespaces and creates a blkdev interface
 * for each namespace found. Namespaces can have various attributes to support
 * thin provisioning and protection information. This driver does not support
 * any of this and ignores namespaces that have these attributes.
 *
 * As of NVMe 1.1 namespaces can have an 64bit Extended Unique Identifier
 * (EUI64). This driver uses the EUI64 if present to generate the devid and
 * passes it to blkdev to use it in the device node names. As this is currently
 * untested namespaces with EUI64 are ignored by default.
 *
 * We currently support only (2 << NVME_MINOR_INST_SHIFT) - 2 namespaces in a
 * single controller. This is an artificial limit imposed by the driver to be
 * able to address a reasonable number of controllers and namespaces using a
 * 32bit minor node number.
 *
 *
 * Minor nodes:
 *
 * For each NVMe device the driver exposes one minor node for the controller and
 * one minor node for each namespace. The only operations supported by those
 * minor nodes are open(9E), close(9E), and ioctl(9E). This serves as the
 * interface for the nvmeadm(1M) utility.
 *
 *
 * Blkdev Interface:
 *
 * This driver uses blkdev to do all the heavy lifting involved with presenting
 * a disk device to the system. As a result, the processing of I/O requests is
 * relatively simple as blkdev takes care of partitioning, boundary checks, DMA
 * setup, and splitting of transfers into manageable chunks.
 *
 * I/O requests coming in from blkdev are turned into NVM commands and posted to
 * an I/O queue. The queue is selected by taking the CPU id modulo the number of
 * queues. There is currently no timeout handling of I/O commands.
 *
```

```
 * Blkdev also supports querying device/media information and generating a
 * devid. The driver reports the best block size as determined by the namespace
 * format back to blkdev as physical block size to support partition and block
 * alignment. The devid is either based on the namespace EUI64, if present, or
 * composed using the device vendor ID, model number, serial number, and the
 * namespace ID.
 *
 *
 * Error Handling:
 *
 * Error handling is currently limited to detecting fatal hardware errors,
 * either by asynchronous events, or synchronously through command status or
 * admin command timeouts. In case of severe errors the device is fenced off,
 * all further requests will return EIO. FMA is then called to fault the device.
 *
 * The hardware has a limit for outstanding asynchronous event requests. Before
 * this limit is known the driver assumes it is at least 1 and posts a single
 * asynchronous request. Later when the limit is known more asynchronous event
 * requests are posted to allow quicker reception of error information. When an
 * asynchronous event is posted by the hardware the driver will parse the error
 * status fields and log information or fault the device, depending on the
 * severity of the asynchronous event. The asynchronous event request is then
 * reused and posted to the admin queue again.
 *
 * On command completion the command status is checked for errors. In case of
 * errors indicating a driver bug the driver panics. Almost all other error
 * status values just cause EIO to be returned.
 *
 * Command timeouts are currently detected for all admin commands except
 * asynchronous event requests. If a command times out and the hardware appears
 * to be healthy the driver attempts to abort the command. The original command
 * timeout is also applied to the abort command. If the abort times out too the
 * to be healthy the driver attempts to abort the command. If this fails the
 * driver assumes the device to be dead, fences it off, and calls FMA to retire
 * it. In all other cases the aborted command should return immediately with a
 * status indicating it was aborted, and the driver will wait indefinitely for
 * that to happen. No timeout handling of normal I/O commands is presently done.
 * it. In general admin commands are issued at attach time only. No timeout
 * handling of normal I/O commands is presently done.
 *
 * Any command that times out due to the controller dropping dead will be put on
 * nvme_lost_cmds list if it references DMA memory. This will prevent the DMA
 * memory being reused by the system and later be written to by a "dead" NVMe
 * controller.
 * In some cases it may be possible that the ABORT command times out, too. In
 * that case the device is also declared dead and fenced off.
 *
 *
 * Locking:
 *
 * Each queue pair has its own nq_mutex, which must be held when accessing the
 * associated queue registers or the shared state of the queue pair. Callers of
 * nvme_unqueue_cmd() must make sure that nq_mutex is held, while
 * nvme_submit_{admin,io}_cmd() and nvme_retrieve_cmd() take care of this
 * themselves.
 *
 * Each command also has its own nc_mutex, which is associated with the
 * condition variable nc_cv. It is only used on admin commands which are run
 * synchronously. In that case it must be held across calls to
 * nvme_submit_{admin,io}_cmd() and nvme_wait_cmd(), which is taken care of by
 * nvme_admin_cmd(). It must also be held whenever the completion state of the
 * command is changed or while a admin command timeout is handled.
 *
 * If both nc_mutex and nq_mutex must be held, nc_mutex must be acquired first.
 * More than one nc_mutex may only be held when aborting commands. In this case,
 * the nc_mutex of the command to be aborted must be held across the call to
```



```
 * nvme_abort_cmd() to prevent the command from completing while the abort is in
 * progress.
 *
 * Each minor node has its own nm_mutex, which protects the open count nm_ocnt
 * and exclusive-open flag nm_oexcl.
 *
 *
 * Quiesce / Fast Reboot:
 *
 * The driver currently does not support fast reboot. A quiesce(9E) entry point
 * is still provided which is used to send a shutdown notification to the
 * device.
 *
 *
 * Driver Configuration:
 *
 * The following driver properties can be changed to control some aspects of the
 * drivers operation:
 * - strict-version: can be set to 0 to allow devices conforming to newer
 *   versions or namespaces with EUI64 to be used
 * - ignore-unknown-vendor-status: can be set to 1 to not handle any vendor
 *   specific command status as a fatal error leading device faulting
 * - admin-queue-len: the maximum length of the admin queue (16-4096)
 * - io-queue-len: the maximum length of the I/O queues (16-65536)
 * - async-event-limit: the maximum number of asynchronous event requests to be
 *   posted by the driver
 * - volatile-write-cache-enable: can be set to 0 to disable the volatile write
 *   cache
 * - min-phys-block-size: the minimum physical block size to report to blkdev,
 *   which is among other things the basis for ZFS vdev ashift
 *
 *
 * TODO:
 * - figure out sane default for I/O queue depth reported to blkdev
 * - FMA handling of media errors
 * - support for devices supporting very large I/O requests using chained PRPs
 * - support for configuring hardware parameters like interrupt coalescing
 * - support for media formatting and hard partitioning into namespaces
 * - support for big-endian systems
 * - support for fast reboot
 * - support for firmware updates
 * - support for NVMe Subsystem Reset (1.1)
 * - support for Scatter/Gather lists (1.1)
 * - support for Reservations (1.1)
 * - support for power management
 */

#include <sys/byteorder.h>
#ifdef _BIG_ENDIAN
#error nvme driver needs porting for big-endian platforms
#endif

#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/devops.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/sunndi.h>
#include <sys/bitmap.h>
#include <sys/sysmacros.h>
#include <sys/param.h>
#include <sys/varargs.h>
#include <sys/cpuvar.h>
#include <sys/disp.h>
#include <sys/blkdev.h>
#include <sys/atomic.h>
```

```
246 #include <sys/archsystm.h>
247 #include <sys/sata/sata_hba.h>
248 #include <sys/stat.h>
249 #include <sys/policy.h>
250 #include <sys/list.h>

252 #include <sys/nvme.h>

254 #ifdef __x86
255 #include <sys/x86_archext.h>
256 #endif

258 #include "nvme_reg.h"
259 #include "nvme_var.h"


262 /* NVMe spec version supported */
263 static const int nvme_version_major = 1;
264 static const int nvme_version_minor = 2;

266 /* tunable for admin command timeout in seconds, default is 1s */
267 int nvme_admin_cmd_timeout = 1;

269 /* tunable for FORMAT NVM command timeout in seconds, default is 600s */
270 int nvme_format_cmd_timeout = 600;

272 static int nvme_attach(dev_info_t *, ddi_attach_cmd_t);
273 static int nvme_detach(dev_info_t *, ddi_detach_cmd_t);
274 static int nvme_quiesce(dev_info_t *);
275 static int nvme_fm_errcb(dev_info_t *, ddi_fm_error_t *, const void *);
276 static int nvme_setup_interrupts(nvme_t *, int, int);
277 static void nvme_release_interrupts(nvme_t *);
278 static uint_t nvme_intr(caddr_t, caddr_t);

280 static void nvme_shutdown(nvme_t *, int, boolean_t);
281 static boolean_t nvme_reset(nvme_t *, boolean_t);
282 static int nvme_init(nvme_t *);
283 static nvme_cmd_t *nvme_alloc_cmd(nvme_t *, int);
284 static void nvme_free_cmd(nvme_cmd_t *);
285 static nvme_cmd_t *nvme_create_nvm_cmd(nvme_namespace_t *, uint8_t,
286     bd_xfer_t *);
287 static void nvme_admin_cmd(nvme_cmd_t *, int);
260 static int nvme_admin_cmd(nvme_cmd_t *, int);
288 static void nvme_submit_admin_cmd(nvme_qpair_t *, nvme_cmd_t *);
289 static int nvme_submit_io_cmd(nvme_qpair_t *, nvme_cmd_t *);
290 static void nvme_submit_cmd_common(nvme_qpair_t *, nvme_cmd_t *);
291 static nvme_cmd_t *nvme_unqueue_cmd(nvme_t *, nvme_qpair_t *, int);
292 static nvme_cmd_t *nvme_retrieve_cmd(nvme_t *, nvme_qpair_t *);
293 static void nvme_wait_cmd(nvme_cmd_t *, uint_t);
265 static boolean_t nvme_wait_cmd(nvme_cmd_t *, uint_t);
294 static void nvme_wakeup_cmd(void *);
295 static void nvme_async_event_task(void *);

297 static int nvme_check_unknown_cmd_status(nvme_cmd_t *);
298 static int nvme_check_vendor_cmd_status(nvme_cmd_t *);
299 static int nvme_check_integrity_cmd_status(nvme_cmd_t *);
300 static int nvme_check_specific_cmd_status(nvme_cmd_t *);
301 static int nvme_check_generic_cmd_status(nvme_cmd_t *);
302 static inline int nvme_check_cmd_status(nvme_cmd_t *);

304 static int nvme_abort_cmd(nvme_cmd_t *, uint_t);
276 static void nvme_abort_cmd(nvme_cmd_t *);
305 static void nvme_async_event(nvme_t *);
306 static int nvme_format_nvm(nvme_t *, uint32_t, uint8_t, boolean_t, uint8_t,
307     boolean_t, uint8_t);
308 static int nvme_get_logpage(nvme_t *, void **, size_t *, uint8_t, ...);
```



```
309 static int nvme_identify(nvme_t *, uint32_t, void **);
310 static int nvme_set_features(nvme_t *, uint32_t, uint8_t, uint32_t,
281 static void *nvme_identify(nvme_t *, uint32_t);
282 static boolean_t nvme_set_features(nvme_t *, uint32_t, uint8_t, uint32_t,
311     uint32_t *);
312 static int nvme_get_features(nvme_t *, uint32_t, uint8_t, uint32_t *,
284 static boolean_t nvme_get_features(nvme_t *, uint32_t, uint8_t, uint32_t *,
313     void **, size_t *);
314 static int nvme_write_cache_set(nvme_t *, boolean_t);
315 static int nvme_set_nqueues(nvme_t *, uint16_t *);
286 static boolean_t nvme_write_cache_set(nvme_t *, boolean_t);
287 static int nvme_set_nqueues(nvme_t *, uint16_t);

317 static void nvme_free_dma(nvme_dma_t *);
318 static int nvme_zalloc_dma(nvme_t *, size_t, uint_t, ddi_dma_attr_t *,
319     nvme_dma_t **);
320 static int nvme_zalloc_queue_dma(nvme_t *, uint32_t, uint16_t, uint_t,
321     nvme_dma_t **);
322 static void nvme_free_qpair(nvme_qpair_t *);
323 static int nvme_alloc_qpair(nvme_t *, uint32_t, nvme_qpair_t **, int);
324 static int nvme_create_io_qpair(nvme_t *, nvme_qpair_t *, uint16_t);

326 static inline void nvme_put64(nvme_t *, uintptr_t, uint64_t);
327 static inline void nvme_put32(nvme_t *, uintptr_t, uint32_t);
328 static inline uint64_t nvme_get64(nvme_t *, uintptr_t);
329 static inline uint32_t nvme_get32(nvme_t *, uintptr_t);

331 static boolean_t nvme_check_regs_hdl(nvme_t *);
332 static boolean_t nvme_check_dma_hdl(nvme_dma_t *);

334 static int nvme_fill_prp(nvme_cmd_t *, bd_xfer_t *);

336 static void nvme_bd_xfer_done(void *);
337 static void nvme_bd_driveinfo(void *, bd_drive_t *);
338 static int nvme_bd_mediainfo(void *, bd_media_t *);
339 static int nvme_bd_cmd(nvme_namespace_t *, bd_xfer_t *, uint8_t);
340 static int nvme_bd_read(void *, bd_xfer_t *);
341 static int nvme_bd_write(void *, bd_xfer_t *);
342 static int nvme_bd_sync(void *, bd_xfer_t *);
343 static int nvme_bd_devid(void *, dev_info_t *, ddi_devid_t *);

345 static int nvme_prp_dma_constructor(void *, void *, int);
346 static void nvme_prp_dma_destructor(void *, void *);

348 static void nvme_prepare_devid(nvme_t *, uint32_t);

350 static int nvme_open(dev_t *, int, int, cred_t *);
351 static int nvme_close(dev_t, int, int, cred_t *);
352 static int nvme_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

354 #define NVME_MINOR_INST_SHIFT   9
355 #define NVME_MINOR(inst, nsid)  (((inst) << NVME_MINOR_INST_SHIFT) | (nsid))
356 #define NVME_MINOR_INST(minor)  ((minor) >> NVME_MINOR_INST_SHIFT)
357 #define NVME_MINOR_NSID(minor)  ((minor) & ((1 << NVME_MINOR_INST_SHIFT) - 1))
358 #define NVME_MINOR_MAX          (NVME_MINOR(1, 0) - 2)

360 static void *nvme_state;
361 static kmem_cache_t *nvme_cmd_cache;

363 /*
364  * DMA attributes for queue DMA memory
365  *
366  * Queue DMA memory must be page aligned. The maximum length of a queue is
367  * 65536 entries, and an entry can be 64 bytes long.
368  */
369 static ddi_dma_attr_t nvme_queue_dma_attr = {
```

```
370         .dma_attr_version       = DMA_ATTR_V0,
371         .dma_attr_addr_lo       = 0,
372         .dma_attr_addr_hi       = 0xffffffffffffffffULL,
373         .dma_attr_count_max     = (UINT16_MAX + 1) * sizeof (nvme_sqe_t) - 1,
374         .dma_attr_align         = 0x1000,
375         .dma_attr_burstsizes    = 0x7ff,
376         .dma_attr_minxfer       = 0x1000,
377         .dma_attr_maxxfer       = (UINT16_MAX + 1) * sizeof (nvme_sqe_t),
378         .dma_attr_seg           = 0xffffffffffffffffULL,
379         .dma_attr_sgllen        = 1,
380         .dma_attr_granular      = 1,
381         .dma_attr_flags         = 0,
382 };
```

*unchanged_portion_omitted_*

```
492 /*
493  * This list will hold commands that have timed out and couldn't be aborted.
494  * As we don't know what the hardware may still do with the DMA memory we can't
495  * free them, so we'll keep them forever on this list where we can easily look
496  * at them with mdb.
497  */
498 static struct list nvme_lost_cmds;
499 static kmutex_t nvme_lc_mutex;

501 int
502 _init(void)
503 {
504         int error;

506         error = ddi_soft_state_init(&nvme_state, sizeof (nvme_t), 1);
507         if (error != DDI_SUCCESS)
508                 return (error);

510         nvme_cmd_cache = kmem_cache_create("nvme_cmd_cache",
511             sizeof (nvme_cmd_t), 64, NULL, NULL, NULL, NULL, NULL, 0);

513         mutex_init(&nvme_lc_mutex, NULL, MUTEX_DRIVER, NULL);
514         list_create(&nvme_lost_cmds, sizeof (nvme_cmd_t),
515             offsetof(nvme_cmd_t, nc_list));

517         bd_mod_init(&nvme_dev_ops);

519         error = mod_install(&nvme_modlinkage);
520         if (error != DDI_SUCCESS) {
521                 ddi_soft_state_fini(&nvme_state);
522                 mutex_destroy(&nvme_lc_mutex);
523                 list_destroy(&nvme_lost_cmds);
524                 bd_mod_fini(&nvme_dev_ops);
525         }

527         return (error);
528 }

530 int
531 _fini(void)
532 {
533         int error;

535         if (!list_is_empty(&nvme_lost_cmds))
536                 return (DDI_FAILURE);

538         error = mod_remove(&nvme_modlinkage);
539         if (error == DDI_SUCCESS) {
540                 ddi_soft_state_fini(&nvme_state);
541                 kmem_cache_destroy(nvme_cmd_cache);
542                 mutex_destroy(&nvme_lc_mutex);
```



```
543                 list_destroy(&nvme_lost_cmds);
544                 bd_mod_fini(&nvme_dev_ops);
545         }

547         return (error);
548 }
```

*unchanged_portion_omitted_*

```
850 static void
851 nvme_free_cmd(nvme_cmd_t *cmd)
852 {
853         /* Don't free commands on the lost commands list. */
854         if (list_link_active(&cmd->nc_list))
855                 return;

857         if (cmd->nc_dma) {
858                 if (cmd->nc_dma->nd_cached)
859                         kmem_cache_free(cmd->nc_nvme->n_prp_cache,
860                             cmd->nc_dma);
861                 else
862                         nvme_free_dma(cmd->nc_dma);
863                 cmd->nc_dma = NULL;
864         }

866         cv_destroy(&cmd->nc_cv);
867         mutex_destroy(&cmd->nc_mutex);

869         kmem_cache_free(nvme_cmd_cache, cmd);
870 }
```

*unchanged_portion_omitted_*

```
922 static nvme_cmd_t *
923 nvme_unqueue_cmd(nvme_t *nvme, nvme_qpair_t *qp, int cid)
924 {
925         nvme_cmd_t *cmd;

927         ASSERT(mutex_owned(&qp->nq_mutex));
928         ASSERT3S(cid, <, qp->nq_nentry);

930         cmd = qp->nq_cmd[cid];
931         qp->nq_cmd[cid] = NULL;
932         ASSERT3U(qp->nq_active_cmds, >, 0);
933         qp->nq_active_cmds--;
934         sema_v(&qp->nq_sema);

936         ASSERT3P(cmd, !=, NULL);
937         ASSERT3P(cmd->nc_nvme, ==, nvme);
938         ASSERT3S(cmd->nc_sqe.sqe_cid, ==, cid);

940         return (cmd);
941 }

943 static nvme_cmd_t *
944 nvme_retrieve_cmd(nvme_t *nvme, nvme_qpair_t *qp)
945 {
946         nvme_reg_cqhdbl_t head = { 0 };

948         nvme_cqe_t *cqe;
949         nvme_cmd_t *cmd;

951         (void) ddi_dma_sync(qp->nq_cqdma->nd_dmah, 0,
952             sizeof (nvme_cqe_t) * qp->nq_nentry, DDI_DMA_SYNC_FORKERNEL);

954         mutex_enter(&qp->nq_mutex);
955         cqe = &qp->nq_cq[qp->nq_cqhead];
```

```
 957         /* Check phase tag of CQE. Hardware inverts it for new entries. */
 958         if (cqe->cqe_sf.sf_p == qp->nq_phase) {
 959                 mutex_exit(&qp->nq_mutex);
 960                 return (NULL);
 961         }

 963         ASSERT(nvme->n_ioq[cqe->cqe_sqid] == qp);
 891         ASSERT(cqe->cqe_cid < qp->nq_nentry);

 965         cmd = nvme_unqueue_cmd(nvme, qp, cqe->cqe_cid);
 893         cmd = qp->nq_cmd[cqe->cqe_cid];
 894         qp->nq_cmd[cqe->cqe_cid] = NULL;
 895         qp->nq_active_cmds--;

 897         ASSERT(cmd != NULL);
 898         ASSERT(cmd->nc_nvme == nvme);
 967         ASSERT(cmd->nc_sqid == cqe->cqe_sqid);
 900         ASSERT(cmd->nc_sqe.sqe_cid == cqe->cqe_cid);
 968         bcopy(cqe, &cmd->nc_cqe, sizeof (nvme_cqe_t));

 970         qp->nq_sqhead = cqe->cqe_sqhd;

 972         head.b.cqhdbl_cqh = qp->nq_cqhead = (qp->nq_cqhead + 1) % qp->nq_nentry;

 974         /* Toggle phase on wrap-around. */
 975         if (qp->nq_cqhead == 0)
 976                 qp->nq_phase = qp->nq_phase ? 0 : 1;

 978         nvme_put32(cmd->nc_nvme, qp->nq_cqhdbl, head.r);
 979         mutex_exit(&qp->nq_mutex);
 913         sema_v(&qp->nq_sema);

 981         return (cmd);
 982 }
_____unchanged_portion_omitted_

1259 static inline int
1260 nvme_check_cmd_status(nvme_cmd_t *cmd)
1261 {
1262         nvme_cqe_t *cqe = &cmd->nc_cqe;

1264         /*
1265          * Take a shortcut if the controller is dead, or if
1266          * command status indicates no error.
1267          */
1268         if (cmd->nc_nvme->n_dead)
1269                 return (EIO);

1198         /* take a shortcut if everything is alright */
1271         if (cqe->cqe_sf.sf_sct == NVME_CQE_SCT_GENERIC &&
1272             cqe->cqe_sf.sf_sc == NVME_CQE_SC_GEN_SUCCESS)
1273                 return (0);

1275         if (cqe->cqe_sf.sf_sct == NVME_CQE_SCT_GENERIC)
1276                 return (nvme_check_generic_cmd_status(cmd));
1277         else if (cqe->cqe_sf.sf_sct == NVME_CQE_SCT_SPECIFIC)
1278                 return (nvme_check_specific_cmd_status(cmd));
1279         else if (cqe->cqe_sf.sf_sct == NVME_CQE_SCT_INTEGRITY)
1280                 return (nvme_check_integrity_cmd_status(cmd));
1281         else if (cqe->cqe_sf.sf_sct == NVME_CQE_SCT_VENDOR)
1282                 return (nvme_check_vendor_cmd_status(cmd));

1284         return (nvme_check_unknown_cmd_status(cmd));
1285 }

1287 static int
```



```
1288 nvme_abort_cmd(nvme_cmd_t *abort_cmd, uint_t sec)
1215 /*
1216  * nvme_abort_cmd_cb -- replaces nc_callback of aborted commands
1217  *
1218  * This functions takes care of cleaning up aborted commands. The command
1219  * status is checked to catch any fatal errors.
1220  */
1221 static void
1222 nvme_abort_cmd_cb(void *arg)
1289 {
1224         nvme_cmd_t *cmd = arg;

1226         /*
1227          * Grab the command mutex. Once we have it we hold the last reference
1228          * to the command and can safely free it.
1229          */
1230         mutex_enter(&cmd->nc_mutex);
1231         (void) nvme_check_cmd_status(cmd);
1232         mutex_exit(&cmd->nc_mutex);

1234         nvme_free_cmd(cmd);
1235 }

1237 static void
1238 nvme_abort_cmd(nvme_cmd_t *abort_cmd)
1239 {
1290         nvme_t *nvme = abort_cmd->nc_nvme;
1291         nvme_cmd_t *cmd = nvme_alloc_cmd(nvme, KM_SLEEP);
1292         nvme_abort_cmd_t ac = { 0 };
1293         int ret = 0;

1295         sema_p(&nvme->n_abort_sema);

1297         ac.b.ac_cid = abort_cmd->nc_sqe.sqe_cid;
1298         ac.b.ac_sqid = abort_cmd->nc_sqid;

1249         /*
1250          * Drop the mutex of the aborted command. From this point on
1251          * we must assume that the abort callback has freed the command.
1252          */
1253         mutex_exit(&abort_cmd->nc_mutex);

1300         cmd->nc_sqid = 0;
1301         cmd->nc_sqe.sqe_opc = NVME_OPC_ABORT;
1302         cmd->nc_callback = nvme_wakeup_cmd;
1303         cmd->nc_sqe.sqe_cdw10 = ac.r;

1305         /*
1306          * Send the ABORT to the hardware. The ABORT command will return _after_
1307          * the aborted command has completed (aborted or otherwise), but since
1308          * we still hold the aborted command's mutex its callback hasn't been
1309          * processed yet.
1262          * the aborted command has completed (aborted or otherwise).
1310          */
1311         nvme_admin_cmd(cmd, sec);
1264         if (nvme_admin_cmd(cmd, nvme_admin_cmd_timeout) != DDI_SUCCESS) {
1312         sema_v(&nvme->n_abort_sema);
1266                 dev_err(nvme->n_dip, CE_WARN,
1267                     "!nvme_admin_cmd failed for ABORT");
1268                 atomic_inc_32(&nvme->n_abort_failed);
1269                 return;
1270         }
1271         sema_v(&nvme->n_abort_sema);

1314         if ((ret = nvme_check_cmd_status(cmd)) != 0) {
1273         if (nvme_check_cmd_status(cmd)) {
```

```
1315                 dev_err(nvme->n_dip, CE_WARN,
1316                     "!ABORT failed with sct = %x, sc = %x",
1317                     cmd->nc_cqe.cqe_sf.sf_sct, cmd->nc_cqe.cqe_sf.sf_sc);
1318                 atomic_inc_32(&nvme->n_abort_failed);
1319         } else {
1320                 dev_err(nvme->n_dip, CE_WARN,
1321                     "!ABORT of command %d/%d %ssuccessful",
1322                     abort_cmd->nc_sqe.sqe_cid, abort_cmd->nc_sqid,
1323                     cmd->nc_cqe.cqe_dw0 & 1 ? "un" : "");
1324                 if ((cmd->nc_cqe.cqe_dw0 & 1) == 0)
1325                         atomic_inc_32(&nvme->n_cmd_aborted);
1326         }

1328         nvme_free_cmd(cmd);
1329         return (ret);
1330 }

1332 /*
1333  * nvme_wait_cmd -- wait for command completion or timeout
1334  *
1288  * Returns B_TRUE if the command completed normally.
1289  *
1290  * Returns B_FALSE if the command timed out and an abort was attempted. The
1291  * command mutex will be dropped and the command must be considered freed. The
1292  * freeing of the command is normally done by the abort command callback.
1293  *
1335  * In case of a serious error or a timeout of the abort command the hardware
1336  * will be declared dead and FMA will be notified.
1337  */
1338 static void
1297 static boolean_t
1339 nvme_wait_cmd(nvme_cmd_t *cmd, uint_t sec)
1340 {
1341         clock_t timeout = ddi_get_lbolt() + drv_usectohz(sec * MICROSEC);
1342         nvme_t *nvme = cmd->nc_nvme;
1343         nvme_reg_csts_t csts;
1344         nvme_qpair_t *qp;

1346         ASSERT(mutex_owned(&cmd->nc_mutex));

1348         while (!cmd->nc_completed) {
1349                 if (cv_timedwait(&cmd->nc_cv, &cmd->nc_mutex, timeout) == -1)
1350                         break;
1351         }

1353         if (cmd->nc_completed)
1354                 return;
1312                 return (B_TRUE);

1356         /*
1357          * The command timed out.
1358          *
1315          * The command timed out. Change the callback to the cleanup function.
1316          */
1317         cmd->nc_callback = nvme_abort_cmd_cb;

1319         /*
1359          * Check controller for fatal status, any errors associated with the
1360          * register or DMA handle, or for a double timeout (abort command timed
1361          * out). If necessary log a warning and call FMA.
1362          */
1363         csts.r = nvme_get32(nvme, NVME_REG_CSTS);
1364         dev_err(nvme->n_dip, CE_WARN, "!command %d/%d timeout, "
1365             "OPC = %x, CFS = %d", cmd->nc_sqe.sqe_cid, cmd->nc_sqid,
1366             cmd->nc_sqe.sqe_opc, csts.b.csts_cfs);
1325         dev_err(nvme->n_dip, CE_WARN, "!command timeout, "
```



```
1326             "OPC = %x, CFS = %d", cmd->nc_sqe.sqe_opc, csts.b.csts_cfs);
1367         atomic_inc_32(&nvme->n_cmd_timeout);

1369         if (csts.b.csts_cfs ||
1370             nvme_check_regs_hdl(nvme) ||
1371             nvme_check_dma_hdl(cmd->nc_dma) ||
1372             cmd->nc_sqe.sqe_opc == NVME_OPC_ABORT) {
1373                 ddi_fm_service_impact(nvme->n_dip, DDI_SERVICE_LOST);
1374                 nvme->n_dead = B_TRUE;
1375         } else if (nvme_abort_cmd(cmd, sec) == 0) {
1335                 mutex_exit(&cmd->nc_mutex);
1336         } else {
1376                 /*
1377                  * If the abort succeeded the command should complete
1378                  * immediately with an appropriate status.
1338                  * Try to abort the command. The command mutex is released by
1339                  * nvme_abort_cmd().
1340                  * If the abort succeeds it will have freed the aborted command.
1341                  * If the abort fails for other reasons we must assume that the
1342                  * command may complete at any time, and the callback will free
1343                  * it for us.
1379                  */
1380                 while (!cmd->nc_completed)
1381                         cv_wait(&cmd->nc_cv, &cmd->nc_mutex);

1383                 return;
1345                 nvme_abort_cmd(cmd);
1384         }

1386         qp = nvme->n_ioq[cmd->nc_sqid];

1388         mutex_enter(&qp->nq_mutex);
1389         (void) nvme_unqueue_cmd(nvme, qp, cmd->nc_sqe.sqe_cid);
1390         mutex_exit(&qp->nq_mutex);

1392         /*
1393          * As we don't know what the presumed dead hardware might still do with
1394          * the DMA memory, we'll put the command on the lost commands list if it
1395          * has any DMA memory.
1396          */
1397         if (cmd->nc_dma != NULL) {
1398                 mutex_enter(&nvme_lc_mutex);
1399                 list_insert_head(&nvme_lost_cmds, cmd);
1400                 mutex_exit(&nvme_lc_mutex);
1401         }
1348         return (B_FALSE);
1402 }

1404 static void
1405 nvme_wakeup_cmd(void *arg)
1406 {
1407         nvme_cmd_t *cmd = arg;

1409         mutex_enter(&cmd->nc_mutex);
1357         /*
1358          * There is a slight chance that this command completed shortly after
1359          * the timeout was hit in nvme_wait_cmd() but before the callback was
1360          * changed. Catch that case here and clean up accordingly.
1361          */
1362         if (cmd->nc_callback == nvme_abort_cmd_cb) {
1363                 mutex_exit(&cmd->nc_mutex);
1364                 nvme_abort_cmd_cb(cmd);
1365                 return;
1366         }

1410         cmd->nc_completed = B_TRUE;
```

```
1411         cv_signal(&cmd->nc_cv);
1412         mutex_exit(&cmd->nc_mutex);
1413 }

1415 static void
1416 nvme_async_event_task(void *arg)
1417 {
1418         nvme_cmd_t *cmd = arg;
1419         nvme_t *nvme = cmd->nc_nvme;
1420         nvme_error_log_entry_t *error_log = NULL;
1421         nvme_health_log_t *health_log = NULL;
1422         size_t logsize = 0;
1423         nvme_async_event_t event;

1425         /*
1426          * Check for errors associated with the async request itself. The only
1427          * command-specific error is "async event limit exceeded", which
1428          * indicates a programming error in the driver and causes a panic in
1429          * nvme_check_cmd_status().
1430          *
1431          * Other possible errors are various scenarios where the async request
1432          * was aborted, or internal errors in the device. Internal errors are
1433          * reported to FMA, the command aborts need no special handling here.
1434          */
1435         if (nvme_check_cmd_status(cmd) != 0) {
1393         if (nvme_check_cmd_status(cmd)) {
1436                 dev_err(cmd->nc_nvme->n_dip, CE_WARN,
1437                     "!async event request returned failure, sct = %x, "
1438                     "sc = %x, dnr = %d, m = %d", cmd->nc_cqe.cqe_sf.sf_sct,
1439                     cmd->nc_cqe.cqe_sf.sf_sc, cmd->nc_cqe.cqe_sf.sf_dnr,
1440                     cmd->nc_cqe.cqe_sf.sf_m);

1442                 if (cmd->nc_cqe.cqe_sf.sf_sct == NVME_CQE_SCT_GENERIC &&
1443                     cmd->nc_cqe.cqe_sf.sf_sc == NVME_CQE_SC_GEN_INTERNAL_ERR) {
1444                         cmd->nc_nvme->n_dead = B_TRUE;
1445                         ddi_fm_service_impact(cmd->nc_nvme->n_dip,
1446                             DDI_SERVICE_LOST);
1447                 }
1448                 nvme_free_cmd(cmd);
1449                 return;
1450         }


1453         event.r = cmd->nc_cqe.cqe_dw0;

1455         /* Clear CQE and re-submit the async request. */
1456         bzero(&cmd->nc_cqe, sizeof (nvme_cqe_t));
1457         nvme_submit_admin_cmd(nvme->n_adminq, cmd);

1459         switch (event.b.ae_type) {
1460         case NVME_ASYNC_TYPE_ERROR:
1461                 if (event.b.ae_logpage == NVME_LOGPAGE_ERROR) {
1462                         (void) nvme_get_logpage(nvme, (void **)&error_log,
1463                             &logsize, event.b.ae_logpage);
1464                 } else {
1465                         dev_err(nvme->n_dip, CE_WARN, "!wrong logpage in "
1466                             "async event reply: %d", event.b.ae_logpage);
1467                         atomic_inc_32(&nvme->n_wrong_logpage);
1468                 }

1470                 switch (event.b.ae_info) {
1471                 case NVME_ASYNC_ERROR_INV_SQ:
1472                         dev_err(nvme->n_dip, CE_PANIC, "programming error: "
1473                             "invalid submission queue");
1474                         return;
```



```
1476                 case NVME_ASYNC_ERROR_INV_DBL:
1477                         dev_err(nvme->n_dip, CE_PANIC, "programming error: "
1478                             "invalid doorbell write value");
1479                         return;

1481                 case NVME_ASYNC_ERROR_DIAGFAIL:
1482                         dev_err(nvme->n_dip, CE_WARN, "!diagnostic failure");
1483                         ddi_fm_service_impact(nvme->n_dip, DDI_SERVICE_LOST);
1484                         nvme->n_dead = B_TRUE;
1485                         atomic_inc_32(&nvme->n_diagfail_event);
1486                         break;

1488                 case NVME_ASYNC_ERROR_PERSISTENT:
1489                         dev_err(nvme->n_dip, CE_WARN, "!persistent internal "
1490                             "device error");
1491                         ddi_fm_service_impact(nvme->n_dip, DDI_SERVICE_LOST);
1492                         nvme->n_dead = B_TRUE;
1493                         atomic_inc_32(&nvme->n_persistent_event);
1494                         break;

1496                 case NVME_ASYNC_ERROR_TRANSIENT:
1497                         dev_err(nvme->n_dip, CE_WARN, "!transient internal "
1498                             "device error");
1499                         /* TODO: send ereport */
1500                         atomic_inc_32(&nvme->n_transient_event);
1501                         break;

1503                 case NVME_ASYNC_ERROR_FW_LOAD:
1504                         dev_err(nvme->n_dip, CE_WARN,
1505                             "!firmware image load error");
1506                         atomic_inc_32(&nvme->n_fw_load_event);
1507                         break;
1508                 }
1509                 break;

1511         case NVME_ASYNC_TYPE_HEALTH:
1512                 if (event.b.ae_logpage == NVME_LOGPAGE_HEALTH) {
1513                         (void) nvme_get_logpage(nvme, (void **)&health_log,
1514                             &logsize, event.b.ae_logpage, -1);
1515                 } else {
1516                         dev_err(nvme->n_dip, CE_WARN, "!wrong logpage in "
1517                             "async event reply: %d", event.b.ae_logpage);
1518                         atomic_inc_32(&nvme->n_wrong_logpage);
1519                 }

1521                 switch (event.b.ae_info) {
1522                 case NVME_ASYNC_HEALTH_RELIABILITY:
1523                         dev_err(nvme->n_dip, CE_WARN,
1524                             "!device reliability compromised");
1525                         /* TODO: send ereport */
1526                         atomic_inc_32(&nvme->n_reliability_event);
1527                         break;

1529                 case NVME_ASYNC_HEALTH_TEMPERATURE:
1530                         dev_err(nvme->n_dip, CE_WARN,
1531                             "!temperature above threshold");
1532                         /* TODO: send ereport */
1533                         atomic_inc_32(&nvme->n_temperature_event);
1534                         break;

1536                 case NVME_ASYNC_HEALTH_SPARE:
1537                         dev_err(nvme->n_dip, CE_WARN,
1538                             "!spare space below threshold");
1539                         /* TODO: send ereport */
1540                         atomic_inc_32(&nvme->n_spare_event);
1541                         break;
```

```
1542                 }
1543                 break;

1545         case NVME_ASYNC_TYPE_VENDOR:
1546                 dev_err(nvme->n_dip, CE_WARN, "!vendor specific async event "
1547                     "received, info = %x, logpage = %x", event.b.ae_info,
1548                     event.b.ae_logpage);
1549                 atomic_inc_32(&nvme->n_vendor_event);
1550                 break;

1552         default:
1553                 dev_err(nvme->n_dip, CE_WARN, "!unknown async event received, "
1554                     "type = %x, info = %x, logpage = %x", event.b.ae_type,
1555                     event.b.ae_info, event.b.ae_logpage);
1556                 atomic_inc_32(&nvme->n_unknown_event);
1557                 break;
1558         }

1560         if (error_log)
1561                 kmem_free(error_log, logsize);

1563         if (health_log)
1564                 kmem_free(health_log, logsize);
1565 }

1567 static void
1525 static int
1568 nvme_admin_cmd(nvme_cmd_t *cmd, int sec)
1569 {
1570         mutex_enter(&cmd->nc_mutex);
1571         nvme_submit_admin_cmd(cmd->nc_nvme->n_adminq, cmd);
1572         nvme_wait_cmd(cmd, sec);

1531         if (nvme_wait_cmd(cmd, sec) == B_FALSE) {
1532                 /*
1533                  * The command timed out. An abort command was posted that
1534                  * will take care of the cleanup.
1535                  */
1536                 return (DDI_FAILURE);
1537         }
1573         mutex_exit(&cmd->nc_mutex);

1540         return (DDI_SUCCESS);
1574 }
______unchanged_portion_omitted_

1588 static int
1589 nvme_format_nvm(nvme_t *nvme, uint32_t nsid, uint8_t lbaf, boolean_t ms,
1590     uint8_t pi, boolean_t pil, uint8_t ses)
1591 {
1592         nvme_cmd_t *cmd = nvme_alloc_cmd(nvme, KM_SLEEP);
1593         nvme_format_nvm_t format_nvm = { 0 };
1594         int ret;

1596         format_nvm.b.fm_lbaf = lbaf & 0xf;
1597         format_nvm.b.fm_ms = ms ? 1 : 0;
1598         format_nvm.b.fm_pi = pi & 0x7;
1599         format_nvm.b.fm_pil = pil ? 1 : 0;
1600         format_nvm.b.fm_ses = ses & 0x7;

1602         cmd->nc_sqid = 0;
1603         cmd->nc_callback = nvme_wakeup_cmd;
1604         cmd->nc_sqe.sqe_nsid = nsid;
1605         cmd->nc_sqe.sqe_opc = NVME_OPC_NVM_FORMAT;
1606         cmd->nc_sqe.sqe_cdw10 = format_nvm.r;
```



```
1608         /*
1609          * Some devices like Samsung SM951 don't allow formatting of all
1610          * namespaces in one command. Handle that gracefully.
1611          */
1612         if (nsid == (uint32_t)-1)
1613                 cmd->nc_dontpanic = B_TRUE;

1615         nvme_admin_cmd(cmd, nvme_format_cmd_timeout);
1582         if ((ret = nvme_admin_cmd(cmd, nvme_format_cmd_timeout))
1583             != DDI_SUCCESS) {
1584                 dev_err(nvme->n_dip, CE_WARN,
1585                     "!nvme_admin_cmd failed for FORMAT NVM");
1586                 return (EIO);
1587         }

1617         if ((ret = nvme_check_cmd_status(cmd)) != 0) {
1618                 dev_err(nvme->n_dip, CE_WARN,
1619                     "!FORMAT failed with sct = %x, sc = %x",
1620                     cmd->nc_cqe.cqe_sf.sf_sct, cmd->nc_cqe.cqe_sf.sf_sc);
1621         }

1623         nvme_free_cmd(cmd);
1624         return (ret);
1625 }

1627 static int
1628 nvme_get_logpage(nvme_t *nvme, void **buf, size_t *bufsize, uint8_t logpage,
1629     ...)
1630 {
1631         nvme_cmd_t *cmd = nvme_alloc_cmd(nvme, KM_SLEEP);
1632         nvme_getlogpage_t getlogpage = { 0 };
1633         va_list ap;
1634         int ret;
1606         int ret = DDI_FAILURE;

1636         va_start(ap, logpage);

1638         cmd->nc_sqid = 0;
1639         cmd->nc_callback = nvme_wakeup_cmd;
1640         cmd->nc_sqe.sqe_opc = NVME_OPC_GET_LOG_PAGE;

1642         getlogpage.b.lp_lid = logpage;

1644         switch (logpage) {
1645         case NVME_LOGPAGE_ERROR:
1646                 cmd->nc_sqe.sqe_nsid = (uint32_t)-1;
1647                 /*
1648                  * The GET LOG PAGE command can use at most 2 pages to return
1649                  * data, PRP lists are not supported.
1650                  */
1651                 *bufsize = MIN(2 * nvme->n_pagesize,
1652                     nvme->n_error_log_len * sizeof (nvme_error_log_entry_t));
1653                 break;

1655         case NVME_LOGPAGE_HEALTH:
1656                 cmd->nc_sqe.sqe_nsid = va_arg(ap, uint32_t);
1657                 *bufsize = sizeof (nvme_health_log_t);
1658                 break;

1660         case NVME_LOGPAGE_FWSLOT:
1661                 cmd->nc_sqe.sqe_nsid = (uint32_t)-1;
1662                 *bufsize = sizeof (nvme_fwslot_log_t);
1663                 break;

1665         default:
1666                 dev_err(nvme->n_dip, CE_WARN, "!unknown log page requested: %d",
```

```
                    logpage);
                atomic_inc_32(&nvme->n_unknown_logpage);
                ret = EINVAL;
                goto fail;
        }

        va_end(ap);

        getlogpage.b.lp_numd = *bufsize / sizeof (uint32_t) - 1;

        cmd->nc_sqe.sqe_cdw10 = getlogpage.r;

        if (nvme_zalloc_dma(nvme, getlogpage.b.lp_numd * sizeof (uint32_t),
            DDI_DMA_READ, &nvme->n_prp_dma_attr, &cmd->nc_dma) != DDI_SUCCESS) {
                dev_err(nvme->n_dip, CE_WARN,
                    "!nvme_zalloc_dma failed for GET LOG PAGE");
                ret = ENOMEM;
                goto fail;
        }

        if (cmd->nc_dma->nd_ncookie > 2) {
                dev_err(nvme->n_dip, CE_WARN,
                    "!too many DMA cookies for GET LOG PAGE");
                atomic_inc_32(&nvme->n_too_many_cookies);
                ret = ENOMEM;
                goto fail;
        }

        cmd->nc_sqe.sqe_dptr.d_prp[0] = cmd->nc_dma->nd_cookie.dmac_laddress;
        if (cmd->nc_dma->nd_ncookie > 1) {
                ddi_dma_nextcookie(cmd->nc_dma->nd_dmah,
                    &cmd->nc_dma->nd_cookie);
                cmd->nc_sqe.sqe_dptr.d_prp[1] =
                    cmd->nc_dma->nd_cookie.dmac_laddress;
        }

        nvme_admin_cmd(cmd, nvme_admin_cmd_timeout);
        if (nvme_admin_cmd(cmd, nvme_admin_cmd_timeout) != DDI_SUCCESS) {
                dev_err(nvme->n_dip, CE_WARN,
                    "!nvme_admin_cmd failed for GET LOG PAGE");
                return (ret);
        }

        if ((ret = nvme_check_cmd_status(cmd)) != 0) {
        if (nvme_check_cmd_status(cmd)) {
                dev_err(nvme->n_dip, CE_WARN,
                    "!GET LOG PAGE failed with sct = %x, sc = %x",
                    cmd->nc_cqe.cqe_sf.sf_sct, cmd->nc_cqe.cqe_sf.sf_sc);
                goto fail;
        }

        *buf = kmem_alloc(*bufsize, KM_SLEEP);
        bcopy(cmd->nc_dma->nd_memp, *buf, *bufsize);

        ret = DDI_SUCCESS;

fail:
        nvme_free_cmd(cmd);

        return (ret);
}

static int
nvme_identify(nvme_t *nvme, uint32_t nsid, void **buf)
static void *
nvme_identify(nvme_t *nvme, uint32_t nsid)
```



```
{
        nvme_cmd_t *cmd = nvme_alloc_cmd(nvme, KM_SLEEP);
        int ret;
        void *buf = NULL;

        if (buf == NULL)
                return (EINVAL);

        cmd->nc_sqid = 0;
        cmd->nc_callback = nvme_wakeup_cmd;
        cmd->nc_sqe.sqe_opc = NVME_OPC_IDENTIFY;
        cmd->nc_sqe.sqe_nsid = nsid;
        cmd->nc_sqe.sqe_cdw10 = nsid ? NVME_IDENTIFY_NSID : NVME_IDENTIFY_CTRL;

        if (nvme_zalloc_dma(nvme, NVME_IDENTIFY_BUFSIZE, DDI_DMA_READ,
            &nvme->n_prp_dma_attr, &cmd->nc_dma) != DDI_SUCCESS) {
                dev_err(nvme->n_dip, CE_WARN,
                    "!nvme_zalloc_dma failed for IDENTIFY");
                ret = ENOMEM;
                goto fail;
        }

        if (cmd->nc_dma->nd_ncookie > 2) {
                dev_err(nvme->n_dip, CE_WARN,
                    "!too many DMA cookies for IDENTIFY");
                atomic_inc_32(&nvme->n_too_many_cookies);
                ret = ENOMEM;
                goto fail;
        }

        cmd->nc_sqe.sqe_dptr.d_prp[0] = cmd->nc_dma->nd_cookie.dmac_laddress;
        if (cmd->nc_dma->nd_ncookie > 1) {
                ddi_dma_nextcookie(cmd->nc_dma->nd_dmah,
                    &cmd->nc_dma->nd_cookie);
                cmd->nc_sqe.sqe_dptr.d_prp[1] =
                    cmd->nc_dma->nd_cookie.dmac_laddress;
        }

        nvme_admin_cmd(cmd, nvme_admin_cmd_timeout);
        if (nvme_admin_cmd(cmd, nvme_admin_cmd_timeout) != DDI_SUCCESS) {
                dev_err(nvme->n_dip, CE_WARN,
                    "!nvme_admin_cmd failed for IDENTIFY");
                return (NULL);
        }

        if ((ret = nvme_check_cmd_status(cmd)) != 0) {
        if (nvme_check_cmd_status(cmd)) {
                dev_err(nvme->n_dip, CE_WARN,
                    "!IDENTIFY failed with sct = %x, sc = %x",
                    cmd->nc_cqe.cqe_sf.sf_sct, cmd->nc_cqe.cqe_sf.sf_sc);
                goto fail;
        }

        *buf = kmem_alloc(NVME_IDENTIFY_BUFSIZE, KM_SLEEP);
        bcopy(cmd->nc_dma->nd_memp, *buf, NVME_IDENTIFY_BUFSIZE);
        buf = kmem_alloc(NVME_IDENTIFY_BUFSIZE, KM_SLEEP);
        bcopy(cmd->nc_dma->nd_memp, buf, NVME_IDENTIFY_BUFSIZE);

fail:
        nvme_free_cmd(cmd);

        return (ret);
        return (buf);
}

static int
```

```
1752 static boolean_t
1779 nvme_set_features(nvme_t *nvme, uint32_t nsid, uint8_t feature, uint32_t val,
1780     uint32_t *res)
1781 {
1782         _NOTE(ARGUNUSED(nsid));
1783         nvme_cmd_t *cmd = nvme_alloc_cmd(nvme, KM_SLEEP);
1784         int ret = EINVAL;
1758         boolean_t ret = B_FALSE;

1786         ASSERT(res != NULL);

1788         cmd->nc_sqid = 0;
1789         cmd->nc_callback = nvme_wakeup_cmd;
1790         cmd->nc_sqe.sqe_opc = NVME_OPC_SET_FEATURES;
1791         cmd->nc_sqe.sqe_cdw10 = feature;
1792         cmd->nc_sqe.sqe_cdw11 = val;

1794         switch (feature) {
1795         case NVME_FEAT_WRITE_CACHE:
1796                 if (!nvme->n_write_cache_present)
1797                         goto fail;
1798                 break;

1800         case NVME_FEAT_NQUEUES:
1801                 break;

1803         default:
1804                 goto fail;
1805         }

1807         nvme_admin_cmd(cmd, nvme_admin_cmd_timeout);
1781         if (nvme_admin_cmd(cmd, nvme_admin_cmd_timeout) != DDI_SUCCESS) {
1782                 dev_err(nvme->n_dip, CE_WARN,
1783                     "!nvme_admin_cmd failed for SET FEATURES");
1784                 return (ret);
1785         }

1809         if ((ret = nvme_check_cmd_status(cmd)) != 0) {
1787         if (nvme_check_cmd_status(cmd)) {
1810                 dev_err(nvme->n_dip, CE_WARN,
1811                     "!SET FEATURES %d failed with sct = %x, sc = %x",
1812                     feature, cmd->nc_cqe.cqe_sf.sf_sct,
1813                     cmd->nc_cqe.cqe_sf.sf_sc);
1814                 goto fail;
1815         }

1817         *res = cmd->nc_cqe.cqe_dw0;
1796         ret = B_TRUE;

1819 fail:
1820         nvme_free_cmd(cmd);
1821         return (ret);
1822 }

1824 static int
1803 static boolean_t
1825 nvme_get_features(nvme_t *nvme, uint32_t nsid, uint8_t feature, uint32_t *res,
1826     void **buf, size_t *bufsize)
1827 {
1828         nvme_cmd_t *cmd = nvme_alloc_cmd(nvme, KM_SLEEP);
1829         int ret = EINVAL;
1808         boolean_t ret = B_FALSE;

1831         ASSERT(res != NULL);

1833         if (bufsize != NULL)
```



```
1834                 *bufsize = 0;

1836         cmd->nc_sqid = 0;
1837         cmd->nc_callback = nvme_wakeup_cmd;
1838         cmd->nc_sqe.sqe_opc = NVME_OPC_GET_FEATURES;
1839         cmd->nc_sqe.sqe_cdw10 = feature;
1840         cmd->nc_sqe.sqe_cdw11 = *res;

1842         switch (feature) {
1843         case NVME_FEAT_ARBITRATION:
1844         case NVME_FEAT_POWER_MGMT:
1845         case NVME_FEAT_TEMPERATURE:
1846         case NVME_FEAT_ERROR:
1847         case NVME_FEAT_NQUEUES:
1848         case NVME_FEAT_INTR_COAL:
1849         case NVME_FEAT_INTR_VECT:
1850         case NVME_FEAT_WRITE_ATOM:
1851         case NVME_FEAT_ASYNC_EVENT:
1852         case NVME_FEAT_PROGRESS:
1853                 break;

1855         case NVME_FEAT_WRITE_CACHE:
1856                 if (!nvme->n_write_cache_present)
1857                         goto fail;
1858                 break;

1860         case NVME_FEAT_LBA_RANGE:
1861                 if (!nvme->n_lba_range_supported)
1862                         goto fail;

1864                 /*
1865                  * The LBA Range Type feature is optional. There doesn't seem
1866                  * be a method of detecting whether it is supported other than
1867                  * using it. This will cause a "invalid field in command" error,
1868                  * which is normally considered a programming error and causes
1869                  * panic in nvme_check_generic_cmd_status().
1870                  */
1871                 cmd->nc_dontpanic = B_TRUE;
1872                 cmd->nc_sqe.sqe_nsid = nsid;
1873                 ASSERT(bufsize != NULL);
1874                 *bufsize = NVME_LBA_RANGE_BUFSIZE;

1876                 break;

1878         case NVME_FEAT_AUTO_PST:
1879                 if (!nvme->n_auto_pst_supported)
1880                         goto fail;

1882                 ASSERT(bufsize != NULL);
1883                 *bufsize = NVME_AUTO_PST_BUFSIZE;
1884                 break;

1886         default:
1887                 goto fail;
1888         }

1890         if (bufsize != NULL && *bufsize != 0) {
1891                 if (nvme_zalloc_dma(nvme, *bufsize, DDI_DMA_READ,
1892                     &nvme->n_prp_dma_attr, &cmd->nc_dma) != DDI_SUCCESS) {
1893                         dev_err(nvme->n_dip, CE_WARN,
1894                             "!nvme_zalloc_dma failed for GET FEATURES");
1895                         ret = ENOMEM;
1896                         goto fail;
1897                 }

1899                 if (cmd->nc_dma->nd_ncookie > 2) {
```

```
                        dev_err(nvme->n_dip, CE_WARN,
                            "!too many DMA cookies for GET FEATURES");
                        atomic_inc_32(&nvme->n_too_many_cookies);
                        ret = ENOMEM;
                        goto fail;
                }

                cmd->nc_sqe.sqe_dptr.d_prp[0] =
                    cmd->nc_dma->nd_cookie.dmac_laddress;
                if (cmd->nc_dma->nd_ncookie > 1) {
                        ddi_dma_nextcookie(cmd->nc_dma->nd_dmah,
                            &cmd->nc_dma->nd_cookie);
                        cmd->nc_sqe.sqe_dptr.d_prp[1] =
                            cmd->nc_dma->nd_cookie.dmac_laddress;
                }
        }

        nvme_admin_cmd(cmd, nvme_admin_cmd_timeout);
        if (nvme_admin_cmd(cmd, nvme_admin_cmd_timeout) != DDI_SUCCESS) {
                dev_err(nvme->n_dip, CE_WARN,
                    "!nvme_admin_cmd failed for GET FEATURES");
                return (ret);
        }

        if ((ret = nvme_check_cmd_status(cmd)) != 0) {
        if (nvme_check_cmd_status(cmd)) {
                if (feature == NVME_FEAT_LBA_RANGE &&
                    cmd->nc_cqe.cqe_sf.sf_sct == NVME_CQE_SCT_GENERIC &&
                    cmd->nc_cqe.cqe_sf.sf_sc == NVME_CQE_SC_GEN_INV_FLD)
                        nvme->n_lba_range_supported = B_FALSE;
                else
                        dev_err(nvme->n_dip, CE_WARN,
                            "!GET FEATURES %d failed with sct = %x, sc = %x",
                            feature, cmd->nc_cqe.cqe_sf.sf_sct,
                            cmd->nc_cqe.cqe_sf.sf_sc);
                goto fail;
        }

        if (bufsize != NULL && *bufsize != 0) {
                ASSERT(buf != NULL);
                *buf = kmem_alloc(*bufsize, KM_SLEEP);
                bcopy(cmd->nc_dma->nd_memp, *buf, *bufsize);
        }

        *res = cmd->nc_cqe.cqe_dw0;
        ret = B_TRUE;

fail:
        nvme_free_cmd(cmd);
        return (ret);
}

static int
static boolean_t
nvme_write_cache_set(nvme_t *nvme, boolean_t enable)
{
        nvme_write_cache_t nwc = { 0 };

        if (enable)
                nwc.b.wc_wce = 1;

        return (nvme_set_features(nvme, 0, NVME_FEAT_WRITE_CACHE, nwc.r,
            &nwc.r));
        if (!nvme_set_features(nvme, 0, NVME_FEAT_WRITE_CACHE, nwc.r, &nwc.r))
                return (B_FALSE);
```



```
        return (B_TRUE);
}

static int
nvme_set_nqueues(nvme_t *nvme, uint16_t *nqueues)
nvme_set_nqueues(nvme_t *nvme, uint16_t nqueues)
{
        nvme_nqueues_t nq = { 0 };
        int ret;

        nq.b.nq_nsq = nq.b.nq_ncq = *nqueues - 1;
        nq.b.nq_nsq = nq.b.nq_ncq = nqueues - 1;

        ret = nvme_set_features(nvme, 0, NVME_FEAT_NQUEUES, nq.r, &nq.r);
        if (!nvme_set_features(nvme, 0, NVME_FEAT_NQUEUES, nq.r, &nq.r)) {
                return (0);
        }

        if (ret == 0) {
                /*
                 * Always use the same number of submission and completion
                 * queues, and never use more than the requested number of
                 * queues.
                 * Always use the same number of submission and completion queues, and
                 * never use more than the requested number of queues.
                 */
                *nqueues = MIN(*nqueues, MIN(nq.b.nq_nsq, nq.b.nq_ncq) + 1);
        }

        return (ret);
        return (MIN(nqueues, MIN(nq.b.nq_nsq, nq.b.nq_ncq) + 1));
}

static int
nvme_create_io_qpair(nvme_t *nvme, nvme_qpair_t *qp, uint16_t idx)
{
        nvme_cmd_t *cmd = nvme_alloc_cmd(nvme, KM_SLEEP);
        nvme_create_queue_dw10_t dw10 = { 0 };
        nvme_create_cq_dw11_t c_dw11 = { 0 };
        nvme_create_sq_dw11_t s_dw11 = { 0 };
        int ret;

        dw10.b.q_qid = idx;
        dw10.b.q_qsize = qp->nq_nentry - 1;

        c_dw11.b.cq_pc = 1;
        c_dw11.b.cq_ien = 1;
        c_dw11.b.cq_iv = idx % nvme->n_intr_cnt;

        cmd->nc_sqid = 0;
        cmd->nc_callback = nvme_wakeup_cmd;
        cmd->nc_sqe.sqe_opc = NVME_OPC_CREATE_CQUEUE;
        cmd->nc_sqe.sqe_cdw10 = dw10.r;
        cmd->nc_sqe.sqe_cdw11 = c_dw11.r;
        cmd->nc_sqe.sqe_dptr.d_prp[0] = qp->nq_cqdma->nd_cookie.dmac_laddress;

        nvme_admin_cmd(cmd, nvme_admin_cmd_timeout);
        if (nvme_admin_cmd(cmd, nvme_admin_cmd_timeout) != DDI_SUCCESS) {
                dev_err(nvme->n_dip, CE_WARN,
                    "!nvme_admin_cmd failed for CREATE CQUEUE");
                return (DDI_FAILURE);
        }

        if ((ret = nvme_check_cmd_status(cmd)) != 0) {
        if (nvme_check_cmd_status(cmd)) {
                dev_err(nvme->n_dip, CE_WARN,
```

```
2006                    "!CREATE CQUEUE failed with sct = %x, sc = %x",
2007                    cmd->nc_cqe.cqe_sf.sf_sct, cmd->nc_cqe.cqe_sf.sf_sc);
2008                goto fail;
1991                nvme_free_cmd(cmd);
1992                return (DDI_FAILURE);
2009        }

2011        nvme_free_cmd(cmd);

2013        s_dw11.b.sq_pc = 1;
2014        s_dw11.b.sq_cqid = idx;

2016        cmd = nvme_alloc_cmd(nvme, KM_SLEEP);
2017        cmd->nc_sqid = 0;
2018        cmd->nc_callback = nvme_wakeup_cmd;
2019        cmd->nc_sqe.sqe_opc = NVME_OPC_CREATE_SQUEUE;
2020        cmd->nc_sqe.sqe_cdw10 = dw10.r;
2021        cmd->nc_sqe.sqe_cdw11 = s_dw11.r;
2022        cmd->nc_sqe.sqe_dptr.d_prp[0] = qp->nq_sqdma->nd_cookie.dmac_laddress;

2024        nvme_admin_cmd(cmd, nvme_admin_cmd_timeout);
2008        if (nvme_admin_cmd(cmd, nvme_admin_cmd_timeout) != DDI_SUCCESS) {
2009                dev_err(nvme->n_dip, CE_WARN,
2010                    "!nvme_admin_cmd failed for CREATE SQUEUE");
2011                return (DDI_FAILURE);
2012        }

2026        if ((ret = nvme_check_cmd_status(cmd)) != 0) {
2014        if (nvme_check_cmd_status(cmd)) {
2027                dev_err(nvme->n_dip, CE_WARN,
2028                    "!CREATE SQUEUE failed with sct = %x, sc = %x",
2029                    cmd->nc_cqe.cqe_sf.sf_sct, cmd->nc_cqe.cqe_sf.sf_sc);
2030                goto fail;
2018                nvme_free_cmd(cmd);
2019                return (DDI_FAILURE);
2031        }

2033 fail:
2034        nvme_free_cmd(cmd);

2036        return (ret);
2024        return (DDI_SUCCESS);
2037 }
_____unchanged_portion_omitted_

2121 static int
2122 nvme_init_ns(nvme_t *nvme, int nsid)
2123 {
2124        nvme_namespace_t *ns = &nvme->n_ns[nsid - 1];
2125        nvme_identify_nsid_t *idns;
2126        int last_rp;

2128        ns->ns_nvme = nvme;
2117        idns = nvme_identify(nvme, nsid);

2130        if (nvme_identify(nvme, nsid, (void **)&idns) != 0) {
2119        if (idns == NULL) {
2131                dev_err(nvme->n_dip, CE_WARN,
2132                    "!failed to identify namespace %d", nsid);
2133                return (DDI_FAILURE);
2134        }

2136        ns->ns_idns = idns;
2137        ns->ns_id = nsid;
2138        ns->ns_block_count = idns->id_nsize;
2139        ns->ns_block_size =
```



```
2140            1 << idns->id_lbaf[idns->id_flbas.lba_format].lbaf_lbads;
2141        ns->ns_best_block_size = ns->ns_block_size;

2143        /*
2144         * Get the EUI64 if present. Use it for devid and device node names.
2145         */
2146        if (NVME_VERSION_ATLEAST(&nvme->n_version, 1, 1))
2147                bcopy(idns->id_eui64, ns->ns_eui64, sizeof (ns->ns_eui64));

2149        /*LINTED: E_BAD_PTR_CAST_ALIGN*/
2150        if (*(uint64_t *)ns->ns_eui64 != 0) {
2151                uint8_t *eui64 = ns->ns_eui64;

2153                (void) snprintf(ns->ns_name, sizeof (ns->ns_name),
2154                    "%02x%02x%02x%02x%02x%02x%02x%02x",
2155                    eui64[0], eui64[1], eui64[2], eui64[3],
2156                    eui64[4], eui64[5], eui64[6], eui64[7]);
2157        } else {
2158                (void) snprintf(ns->ns_name, sizeof (ns->ns_name), "%d",
2159                    ns->ns_id);

2161                nvme_prepare_devid(nvme, ns->ns_id);
2162        }

2164        /*
2165         * Find the LBA format with no metadata and the best relative
2166         * performance. A value of 3 means "degraded", 0 is best.
2167         */
2168        last_rp = 3;
2169        for (int j = 0; j <= idns->id_nlbaf; j++) {
2170                if (idns->id_lbaf[j].lbaf_lbads == 0)
2171                        break;
2172                if (idns->id_lbaf[j].lbaf_ms != 0)
2173                        continue;
2174                if (idns->id_lbaf[j].lbaf_rp >= last_rp)
2175                        continue;
2176                last_rp = idns->id_lbaf[j].lbaf_rp;
2177                ns->ns_best_block_size =
2178                    1 << idns->id_lbaf[j].lbaf_lbads;
2179        }

2181        if (ns->ns_best_block_size < nvme->n_min_block_size)
2182                ns->ns_best_block_size = nvme->n_min_block_size;

2184        /*
2185         * We currently don't support namespaces that use either:
2186         * - thin provisioning
2187         * - protection information
2188         * - illegal block size (< 512)
2189         */
2190        if (idns->id_nsfeat.f_thin ||
2191            idns->id_dps.dp_pinfo) {
2192                dev_err(nvme->n_dip, CE_WARN,
2193                    "!ignoring namespace %d, unsupported features: "
2194                    "thin = %d, pinfo = %d", nsid,
2195                    idns->id_nsfeat.f_thin, idns->id_dps.dp_pinfo);
2196                ns->ns_ignore = B_TRUE;
2197        } else if (ns->ns_block_size < 512) {
2198                dev_err(nvme->n_dip, CE_WARN,
2199                    "!ignoring namespace %d, unsupported block size %"PRIu64,
2200                    nsid, (uint64_t)ns->ns_block_size);
2201                ns->ns_ignore = B_TRUE;
2202        } else {
2203                ns->ns_ignore = B_FALSE;
2204        }
```

```
2206         return (DDI_SUCCESS);
2207 }

2209 static int
2210 nvme_init(nvme_t *nvme)
2211 {
2212         nvme_reg_cc_t cc = { 0 };
2213         nvme_reg_aqa_t aqa = { 0 };
2214         nvme_reg_asq_t asq = { 0 };
2215         nvme_reg_acq_t acq = { 0 };
2216         nvme_reg_cap_t cap;
2217         nvme_reg_vs_t vs;
2218         nvme_reg_csts_t csts;
2219         int i = 0;
2220         uint16_t nqueues;
2209         int nqueues;
2221         char model[sizeof (nvme->n_idctl->id_model) + 1];
2222         char *vendor, *product;

2224         /* Check controller version */
2225         vs.r = nvme_get32(nvme, NVME_REG_VS);
2226         nvme->n_version.v_major = vs.b.vs_mjr;
2227         nvme->n_version.v_minor = vs.b.vs_mnr;
2228         dev_err(nvme->n_dip, CE_CONT, "?NVMe spec version %d.%d",
2229             nvme->n_version.v_major, nvme->n_version.v_minor);

2231         if (NVME_VERSION_HIGHER(&nvme->n_version,
2232             nvme_version_major, nvme_version_minor)) {
2233                 dev_err(nvme->n_dip, CE_WARN, "!no support for version > %d.%d",
2234                     nvme_version_major, nvme_version_minor);
2235                 if (nvme->n_strict_version)
2236                         goto fail;
2237         }

2239         /* retrieve controller configuration */
2240         cap.r = nvme_get64(nvme, NVME_REG_CAP);

2242         if ((cap.b.cap_css & NVME_CAP_CSS_NVM) == 0) {
2243                 dev_err(nvme->n_dip, CE_WARN,
2244                     "!NVM command set not supported by hardware");
2245                 goto fail;
2246         }

2248         nvme->n_nssr_supported = cap.b.cap_nssrs;
2249         nvme->n_doorbell_stride = 4 << cap.b.cap_dstrd;
2250         nvme->n_timeout = cap.b.cap_to;
2251         nvme->n_arbitration_mechanisms = cap.b.cap_ams;
2252         nvme->n_cont_queues_reqd = cap.b.cap_cqr;
2253         nvme->n_max_queue_entries = cap.b.cap_mqes + 1;

2255         /*
2256          * The MPSMIN and MPSMAX fields in the CAP register use 0 to specify
2257          * the base page size of 4k (1<<12), so add 12 here to get the real
2258          * page size value.
2259          */
2260         nvme->n_pageshift = MIN(MAX(cap.b.cap_mpsmin + 12, PAGESHIFT),
2261             cap.b.cap_mpsmax + 12);
2262         nvme->n_pagesize = 1UL << (nvme->n_pageshift);

2264         /*
2265          * Set up Queue DMA to transfer at least 1 page-aligned page at a time.
2266          */
2267         nvme->n_queue_dma_attr.dma_attr_align = nvme->n_pagesize;
2268         nvme->n_queue_dma_attr.dma_attr_minxfer = nvme->n_pagesize;

2270         /*
```



```
2271          * Set up PRP DMA to transfer 1 page-aligned page at a time.
2272          * Maxxfer may be increased after we identified the controller limits.
2273          */
2274         nvme->n_prp_dma_attr.dma_attr_maxxfer = nvme->n_pagesize;
2275         nvme->n_prp_dma_attr.dma_attr_minxfer = nvme->n_pagesize;
2276         nvme->n_prp_dma_attr.dma_attr_align = nvme->n_pagesize;
2277         nvme->n_prp_dma_attr.dma_attr_seg = nvme->n_pagesize - 1;

2279         /*
2280          * Reset controller if it's still in ready state.
2281          */
2282         if (nvme_reset(nvme, B_FALSE) == B_FALSE) {
2283                 dev_err(nvme->n_dip, CE_WARN, "!unable to reset controller");
2284                 ddi_fm_service_impact(nvme->n_dip, DDI_SERVICE_LOST);
2285                 nvme->n_dead = B_TRUE;
2286                 goto fail;
2287         }

2289         /*
2290          * Create the admin queue pair.
2291          */
2292         if (nvme_alloc_qpair(nvme, nvme->n_admin_queue_len, &nvme->n_adminq, 0)
2293             != DDI_SUCCESS) {
2294                 dev_err(nvme->n_dip, CE_WARN,
2295                     "!unable to allocate admin qpair");
2296                 goto fail;
2297         }
2298         nvme->n_ioq = kmem_alloc(sizeof (nvme_qpair_t *), KM_SLEEP);
2299         nvme->n_ioq[0] = nvme->n_adminq;

2301         nvme->n_progress |= NVME_ADMIN_QUEUE;

2303         (void) ddi_prop_update_int(DDI_DEV_T_NONE, nvme->n_dip,
2304             "admin-queue-len", nvme->n_admin_queue_len);

2306         aqa.b.aqa_asqs = aqa.b.aqa_acqs = nvme->n_admin_queue_len - 1;
2307         asq = nvme->n_adminq->nq_sqdma->nd_cookie.dmac_laddress;
2308         acq = nvme->n_adminq->nq_cqdma->nd_cookie.dmac_laddress;

2310         ASSERT((asq & (nvme->n_pagesize - 1)) == 0);
2311         ASSERT((acq & (nvme->n_pagesize - 1)) == 0);

2313         nvme_put32(nvme, NVME_REG_AQA, aqa.r);
2314         nvme_put64(nvme, NVME_REG_ASQ, asq);
2315         nvme_put64(nvme, NVME_REG_ACQ, acq);

2317         cc.b.cc_ams = 0;        /* use Round-Robin arbitration */
2318         cc.b.cc_css = 0;        /* use NVM command set */
2319         cc.b.cc_mps = nvme->n_pageshift - 12;
2320         cc.b.cc_shn = 0;        /* no shutdown in progress */
2321         cc.b.cc_en = 1;         /* enable controller */
2322         cc.b.cc_iosqes = 6;     /* submission queue entry is 2^6 bytes long */
2323         cc.b.cc_iocqes = 4;     /* completion queue entry is 2^4 bytes long */

2325         nvme_put32(nvme, NVME_REG_CC, cc.r);

2327         /*
2328          * Wait for the controller to become ready.
2329          */
2330         csts.r = nvme_get32(nvme, NVME_REG_CSTS);
2331         if (csts.b.csts_rdy == 0) {
2332                 for (i = 0; i != nvme->n_timeout * 10; i++) {
2333                         delay(drv_usectohz(50000));
2334                         csts.r = nvme_get32(nvme, NVME_REG_CSTS);

2336                         if (csts.b.csts_cfs == 1) {
```

```
2337                                dev_err(nvme->n_dip, CE_WARN,
2338                                    "!controller fatal status at init");
2339                                ddi_fm_service_impact(nvme->n_dip,
2340                                    DDI_SERVICE_LOST);
2341                                nvme->n_dead = B_TRUE;
2342                                goto fail;
2343                        }

2345                        if (csts.b.csts_rdy == 1)
2346                                break;
2347                }
2348        }

2350        if (csts.b.csts_rdy == 0) {
2351                dev_err(nvme->n_dip, CE_WARN, "!controller not ready");
2352                ddi_fm_service_impact(nvme->n_dip, DDI_SERVICE_LOST);
2353                nvme->n_dead = B_TRUE;
2354                goto fail;
2355        }

2357        /*
2358         * Assume an abort command limit of 1. We'll destroy and re-init
2359         * that later when we know the true abort command limit.
2360         */
2361        sema_init(&nvme->n_abort_sema, 1, NULL, SEMA_DRIVER, NULL);

2363        /*
2364         * Setup initial interrupt for admin queue.
2365         */
2366        if ((nvme_setup_interrupts(nvme, DDI_INTR_TYPE_MSIX, 1)
2367            != DDI_SUCCESS) &&
2368            (nvme_setup_interrupts(nvme, DDI_INTR_TYPE_MSI, 1)
2369            != DDI_SUCCESS) &&
2370            (nvme_setup_interrupts(nvme, DDI_INTR_TYPE_FIXED, 1)
2371            != DDI_SUCCESS)) {
2372                dev_err(nvme->n_dip, CE_WARN,
2373                    "!failed to setup initial interrupt");
2374                goto fail;
2375        }

2377        /*
2378         * Post an asynchronous event command to catch errors.
2379         */
2380        nvme_async_event(nvme);

2382        /*
2383         * Identify Controller
2384         */
2385        if (nvme_identify(nvme, 0, (void **)&nvme->n_idctl) != 0) {
2374        nvme->n_idctl = nvme_identify(nvme, 0);
2375        if (nvme->n_idctl == NULL) {
2386                dev_err(nvme->n_dip, CE_WARN,
2387                    "!failed to identify controller");
2388                goto fail;
2389        }

2391        /*
2392         * Get Vendor & Product ID
2393         */
2394        bcopy(nvme->n_idctl->id_model, model, sizeof (nvme->n_idctl->id_model));
2395        model[sizeof (nvme->n_idctl->id_model)] = '\0';
2396        sata_split_model(model, &vendor, &product);

2398        if (vendor == NULL)
2399                nvme->n_vendor = strdup("NVMe");
2400        else
```



```
2401                nvme->n_vendor = strdup(vendor);

2403        nvme->n_product = strdup(product);

2405        /*
2406         * Get controller limits.
2407         */
2408        nvme->n_async_event_limit = MAX(NVME_MIN_ASYNC_EVENT_LIMIT,
2409            MIN(nvme->n_admin_queue_len / 10,
2410            MIN(nvme->n_idctl->id_aerl + 1, nvme->n_async_event_limit)));

2412        (void) ddi_prop_update_int(DDI_DEV_T_NONE, nvme->n_dip,
2413            "async-event-limit", nvme->n_async_event_limit);

2415        nvme->n_abort_command_limit = nvme->n_idctl->id_acl + 1;

2417        /*
2418         * Reinitialize the semaphore with the true abort command limit
2419         * supported by the hardware. It's not necessary to disable interrupts
2420         * as only command aborts use the semaphore, and no commands are
2421         * executed or aborted while we're here.
2422         */
2423        sema_destroy(&nvme->n_abort_sema);
2424        sema_init(&nvme->n_abort_sema, nvme->n_abort_command_limit - 1, NULL,
2425            SEMA_DRIVER, NULL);

2427        nvme->n_progress |= NVME_CTRL_LIMITS;

2429        if (nvme->n_idctl->id_mdts == 0)
2430                nvme->n_max_data_transfer_size = nvme->n_pagesize * 65536;
2431        else
2432                nvme->n_max_data_transfer_size =
2433                    1ull << (nvme->n_pageshift + nvme->n_idctl->id_mdts);

2435        nvme->n_error_log_len = nvme->n_idctl->id_elpe + 1;

2437        /*
2438         * Limit n_max_data_transfer_size to what we can handle in one PRP.
2439         * Chained PRPs are currently unsupported.
2440         *
2441         * This is a no-op on hardware which doesn't support a transfer size
2442         * big enough to require chained PRPs.
2443         */
2444        nvme->n_max_data_transfer_size = MIN(nvme->n_max_data_transfer_size,
2445            (nvme->n_pagesize / sizeof (uint64_t) * nvme->n_pagesize));

2447        nvme->n_prp_dma_attr.dma_attr_maxxfer = nvme->n_max_data_transfer_size;

2449        /*
2450         * Make sure the minimum/maximum queue entry sizes are not
2451         * larger/smaller than the default.
2452         */

2454        if (((1 << nvme->n_idctl->id_sqes.qes_min) > sizeof (nvme_sqe_t)) ||
2455            ((1 << nvme->n_idctl->id_sqes.qes_max) < sizeof (nvme_sqe_t)) ||
2456            ((1 << nvme->n_idctl->id_cqes.qes_min) > sizeof (nvme_cqe_t)) ||
2457            ((1 << nvme->n_idctl->id_cqes.qes_max) < sizeof (nvme_cqe_t)))
2458                goto fail;

2460        /*
2461         * Check for the presence of a Volatile Write Cache. If present,
2462         * enable or disable based on the value of the property
2463         * volatile-write-cache-enable (default is enabled).
2464         */
2465        nvme->n_write_cache_present =
2466            nvme->n_idctl->id_vwc.vwc_present == 0 ? B_FALSE : B_TRUE;
```

```
2468         (void) ddi_prop_update_int(DDI_DEV_T_NONE, nvme->n_dip,
2469             "volatile-write-cache-present",
2470             nvme->n_write_cache_present ? 1 : 0);

2472         if (!nvme->n_write_cache_present) {
2473                 nvme->n_write_cache_enabled = B_FALSE;
2474         } else if (nvme_write_cache_set(nvme, nvme->n_write_cache_enabled)
2475             != 0) {
2464         } else if (!nvme_write_cache_set(nvme, nvme->n_write_cache_enabled)) {
2476                 dev_err(nvme->n_dip, CE_WARN,
2477                     "!failed to %sable volatile write cache",
2478                     nvme->n_write_cache_enabled ? "en" : "dis");
2479                 /*
2480                  * Assume the cache is (still) enabled.
2481                  */
2482                 nvme->n_write_cache_enabled = B_TRUE;
2483         }

2485         (void) ddi_prop_update_int(DDI_DEV_T_NONE, nvme->n_dip,
2486             "volatile-write-cache-enable",
2487             nvme->n_write_cache_enabled ? 1 : 0);

2489         /*
2490          * Assume LBA Range Type feature is supported. If it isn't this
2491          * will be set to B_FALSE by nvme_get_features().
2492          */
2493         nvme->n_lba_range_supported = B_TRUE;

2495         /*
2496          * Check support for Autonomous Power State Transition.
2497          */
2498         if (NVME_VERSION_ATLEAST(&nvme->n_version, 1, 1))
2499                 nvme->n_auto_pst_supported =
2500                     nvme->n_idctl->id_apsta.ap_sup == 0 ? B_FALSE : B_TRUE;

2502         /*
2503          * Identify Namespaces
2504          */
2505         nvme->n_namespace_count = nvme->n_idctl->id_nn;
2506         if (nvme->n_namespace_count > NVME_MINOR_MAX) {
2507                 dev_err(nvme->n_dip, CE_WARN,
2508                     "!too many namespaces: %d, limiting to %d\n",
2509                     nvme->n_namespace_count, NVME_MINOR_MAX);
2510                 nvme->n_namespace_count = NVME_MINOR_MAX;
2511         }

2513         nvme->n_ns = kmem_zalloc(sizeof (nvme_namespace_t) *
2514             nvme->n_namespace_count, KM_SLEEP);

2516         for (i = 0; i != nvme->n_namespace_count; i++) {
2517                 mutex_init(&nvme->n_ns[i].ns_minor.nm_mutex, NULL, MUTEX_DRIVER,
2518                     NULL);
2519                 if (nvme_init_ns(nvme, i + 1) != DDI_SUCCESS)
2520                         goto fail;
2521         }

2523         /*
2524          * Try to set up MSI/MSI-X interrupts.
2525          */
2526         if ((nvme->n_intr_types & (DDI_INTR_TYPE_MSI | DDI_INTR_TYPE_MSIX))
2527             != 0) {
2528                 nvme_release_interrupts(nvme);

2530                 nqueues = MIN(UINT16_MAX, ncpus);
```



```
2532                 if ((nvme_setup_interrupts(nvme, DDI_INTR_TYPE_MSIX,
2533                     nqueues) != DDI_SUCCESS) &&
2534                     (nvme_setup_interrupts(nvme, DDI_INTR_TYPE_MSI,
2535                     nqueues) != DDI_SUCCESS)) {
2536                         dev_err(nvme->n_dip, CE_WARN,
2537                             "!failed to setup MSI/MSI-X interrupts");
2538                         goto fail;
2539                 }
2540         }

2542         nqueues = nvme->n_intr_cnt;

2544         /*
2545          * Create I/O queue pairs.
2546          */

2548         if (nvme_set_nqueues(nvme, &nqueues) != 0) {
2536         nvme->n_ioq_count = nvme_set_nqueues(nvme, nqueues);
2537         if (nvme->n_ioq_count == 0) {
2549                 dev_err(nvme->n_dip, CE_WARN,
2550                     "!failed to set number of I/O queues to %d",
2551                     nvme->n_intr_cnt);
2539                     "!failed to set number of I/O queues to %d", nqueues);
2552                 goto fail;
2553         }

2555         /*
2556          * Reallocate I/O queue array
2557          */
2558         kmem_free(nvme->n_ioq, sizeof (nvme_qpair_t *));
2559         nvme->n_ioq = kmem_zalloc(sizeof (nvme_qpair_t *) *
2560             (nqueues + 1), KM_SLEEP);
2548             (nvme->n_ioq_count + 1), KM_SLEEP);
2561         nvme->n_ioq[0] = nvme->n_adminq;

2563         nvme->n_ioq_count = nqueues;

2565         /*
2566          * If we got less queues than we asked for we might as well give
2567          * some of the interrupt vectors back to the system.
2568          */
2569         if (nvme->n_ioq_count < nvme->n_intr_cnt) {
2555         if (nvme->n_ioq_count < nqueues) {
2570                 nvme_release_interrupts(nvme);

2572                 if (nvme_setup_interrupts(nvme, nvme->n_intr_type,
2573                     nvme->n_ioq_count) != DDI_SUCCESS) {
2574                         dev_err(nvme->n_dip, CE_WARN,
2575                             "!failed to reduce number of interrupts");
2576                         goto fail;
2577                 }
2578         }

2580         /*
2581          * Alloc & register I/O queue pairs
2582          */
2583         nvme->n_io_queue_len =
2584             MIN(nvme->n_io_queue_len, nvme->n_max_queue_entries);
2585         (void) ddi_prop_update_int(DDI_DEV_T_NONE, nvme->n_dip, "io-queue-len",
2586             nvme->n_io_queue_len);

2588         for (i = 1; i != nvme->n_ioq_count + 1; i++) {
2589                 if (nvme_alloc_qpair(nvme, nvme->n_io_queue_len,
2590                     &nvme->n_ioq[i], i) != DDI_SUCCESS) {
2591                         dev_err(nvme->n_dip, CE_WARN,
2592                             "!unable to allocate I/O qpair %d", i);
```

```
2593                        goto fail;
2594                }

2596                if (nvme_create_io_qpair(nvme, nvme->n_ioq[i], i) != 0) {
2582                if (nvme_create_io_qpair(nvme, nvme->n_ioq[i], i)
2583                    != DDI_SUCCESS) {
2597                        dev_err(nvme->n_dip, CE_WARN,
2598                            "!unable to create I/O qpair %d", i);
2599                        goto fail;
2600                }
2601        }

2603        /*
2604         * Post more asynchronous events commands to reduce event reporting
2605         * latency as suggested by the spec.
2606         */
2607        for (i = 1; i != nvme->n_async_event_limit; i++)
2608                nvme_async_event(nvme);

2610        return (DDI_SUCCESS);

2612 fail:
2613        (void) nvme_reset(nvme, B_FALSE);
2614        return (DDI_FAILURE);
2615 }

2617 static uint_t
2618 nvme_intr(caddr_t arg1, caddr_t arg2)
2619 {
2620        /*LINTED: E_PTR_BAD_CAST_ALIGN*/
2621        nvme_t *nvme = (nvme_t *)arg1;
2622        int inum = (int)(uintptr_t)arg2;
2623        int ccnt = 0;
2624        int qnum;
2625        nvme_cmd_t *cmd;

2627        if (inum >= nvme->n_intr_cnt)
2628                return (DDI_INTR_UNCLAIMED);

2630        if (nvme->n_dead)
2631                return (nvme->n_intr_type == DDI_INTR_TYPE_FIXED ?
2632                    DDI_INTR_UNCLAIMED : DDI_INTR_CLAIMED);

2634        /*
2635         * The interrupt vector a queue uses is calculated as queue_idx %
2636         * intr_cnt in nvme_create_io_qpair(). Iterate through the queue array
2637         * in steps of n_intr_cnt to process all queues using this vector.
2638         */
2639        for (qnum = inum;
2640            qnum < nvme->n_ioq_count + 1 && nvme->n_ioq[qnum] != NULL;
2641            qnum += nvme->n_intr_cnt) {
2642                while ((cmd = nvme_retrieve_cmd(nvme, nvme->n_ioq[qnum]))) {
2643                        taskq_dispatch_ent((taskq_t *)cmd->nc_nvme->n_cmd_taskq,
2644                            cmd->nc_callback, cmd, TQ_NOSLEEP, &cmd->nc_tqent);
2645                        ccnt++;
2646                }
2647        }

2649        return (ccnt > 0 ? DDI_INTR_CLAIMED : DDI_INTR_UNCLAIMED);
2650 }
```

*_____unchanged_portion_omitted_*

```
3374 static int
3375 nvme_open(dev_t *devp, int flag, int otyp, cred_t *cred_p)
3376 {
3377 #ifndef __lock_lint
```



```
3378        _NOTE(ARGUNUSED(cred_p));
3379 #endif
3380        minor_t minor = getminor(*devp);
3381        nvme_t *nvme = ddi_get_soft_state(nvme_state, NVME_MINOR_INST(minor));
3382        int nsid = NVME_MINOR_NSID(minor);
3383        nvme_minor_state_t *nm;
3384        int rv = 0;

3386        if (otyp != OTYP_CHR)
3387                return (EINVAL);

3389        if (nvme == NULL)
3390                return (ENXIO);

3392        if (nsid > nvme->n_namespace_count)
3393                return (ENXIO);

3395        if (nvme->n_dead)
3396                return (EIO);

3398        nm = nsid == 0 ? &nvme->n_minor : &nvme->n_ns[nsid - 1].ns_minor;

3400        mutex_enter(&nm->nm_mutex);
3401        if (nm->nm_oexcl) {
3402                rv = EBUSY;
3403                goto out;
3404        }

3406        if (flag & FEXCL) {
3407                if (nm->nm_ocnt != 0) {
3408                        rv = EBUSY;
3409                        goto out;
3410                }
3411                nm->nm_oexcl = B_TRUE;
3412        }

3414        nm->nm_ocnt++;

3416 out:
3417        mutex_exit(&nm->nm_mutex);
3418        return (rv);

3420 }
```

*_____unchanged_portion_omitted_*

```
3456 static int
3457 nvme_ioctl_identify(nvme_t *nvme, int nsid, nvme_ioctl_t *nioc, int mode,
3458     cred_t *cred_p)
3459 {
3460        _NOTE(ARGUNUSED(cred_p));
3461        int rv = 0;
3462        void *idctl;

3464        if ((mode & FREAD) == 0)
3465                return (EPERM);

3467        if (nioc->n_len < NVME_IDENTIFY_BUFSIZE)
3468                return (EINVAL);

3470        if ((rv = nvme_identify(nvme, nsid, (void **)&idctl)) != 0)
3471                return (rv);
3450        idctl = nvme_identify(nvme, nsid);
3451        if (idctl == NULL)
3452                return (EIO);

3473        if (ddi_copyout(idctl, (void *)nioc->n_buf, NVME_IDENTIFY_BUFSIZE, mode)
```

```
3474             != 0)
3475                 rv = EFAULT;

3477         kmem_free(idctl, NVME_IDENTIFY_BUFSIZE);

3479         return (rv);
3480 }
_____unchanged_portion_omitted_

3564 static int
3565 nvme_ioctl_get_features(nvme_t *nvme, int nsid, nvme_ioctl_t *nioc,
3566     int mode, cred_t *cred_p)
3567 {
3568         _NOTE(ARGUNUSED(cred_p));
3569         void *buf = NULL;
3570         size_t bufsize = 0;
3571         uint32_t res = 0;
3572         uint8_t feature;
3573         int rv = 0;

3575         if ((mode & FREAD) == 0)
3576                 return (EPERM);

3578         if ((nioc->n_arg >> 32) > 0xff)
3579                 return (EINVAL);

3581         feature = (uint8_t)(nioc->n_arg >> 32);

3583         switch (feature) {
3584         case NVME_FEAT_ARBITRATION:
3585         case NVME_FEAT_POWER_MGMT:
3586         case NVME_FEAT_TEMPERATURE:
3587         case NVME_FEAT_ERROR:
3588         case NVME_FEAT_NQUEUES:
3589         case NVME_FEAT_INTR_COAL:
3590         case NVME_FEAT_WRITE_ATOM:
3591         case NVME_FEAT_ASYNC_EVENT:
3592         case NVME_FEAT_PROGRESS:
3593                 if (nsid != 0)
3594                         return (EINVAL);
3595                 break;

3597         case NVME_FEAT_INTR_VECT:
3598                 if (nsid != 0)
3599                         return (EINVAL);

3601                 res = nioc->n_arg & 0xffffffffUL;
3602                 if (res >= nvme->n_intr_cnt)
3603                         return (EINVAL);
3604                 break;

3606         case NVME_FEAT_LBA_RANGE:
3607                 if (nvme->n_lba_range_supported == B_FALSE)
3608                         return (EINVAL);

3610                 if (nsid == 0 ||
3611                     nsid > nvme->n_namespace_count)
3612                         return (EINVAL);

3614                 break;

3616         case NVME_FEAT_WRITE_CACHE:
3617                 if (nsid != 0)
3618                         return (EINVAL);

3620                 if (!nvme->n_write_cache_present)
```



```
3621                         return (EINVAL);

3623                 break;

3625         case NVME_FEAT_AUTO_PST:
3626                 if (nsid != 0)
3627                         return (EINVAL);

3629                 if (!nvme->n_auto_pst_supported)
3630                         return (EINVAL);

3632                 break;

3634         default:
3635                 return (EINVAL);
3636         }

3638         rv = nvme_get_features(nvme, nsid, feature, &res, &buf, &bufsize);
3639         if (rv != 0)
3640                 return (rv);
3619         if (nvme_get_features(nvme, nsid, feature, &res, &buf, &bufsize) ==
3620             B_FALSE)
3621                 return (EIO);

3642         if (nioc->n_len < bufsize) {
3643                 kmem_free(buf, bufsize);
3644                 return (EINVAL);
3645         }

3647         if (buf && ddi_copyout(buf, (void*)nioc->n_buf, bufsize, mode) != 0)
3648                 rv = EFAULT;

3650         kmem_free(buf, bufsize);
3651         nioc->n_arg = res;
3652         nioc->n_len = bufsize;

3654         return (rv);
3655 }
_____unchanged_portion_omitted_

3799 static int
3800 nvme_ioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *cred_p,
3801     int *rval_p)
3802 {
3803 #ifndef __lock_lint
3804         _NOTE(ARGUNUSED(rval_p));
3805 #endif
3806         minor_t minor = getminor(dev);
3807         nvme_t *nvme = ddi_get_soft_state(nvme_state, NVME_MINOR_INST(minor));
3808         int nsid = NVME_MINOR_NSID(minor);
3809         int rv = 0;
3810         nvme_ioctl_t nioc;

3812         int (*nvme_ioctl[])(nvme_t *, int, nvme_ioctl_t *, int, cred_t *) = {
3813                 NULL,
3814                 nvme_ioctl_identify,
3815                 nvme_ioctl_identify,
3816                 nvme_ioctl_capabilities,
3817                 nvme_ioctl_get_logpage,
3818                 nvme_ioctl_get_features,
3819                 nvme_ioctl_intr_cnt,
3820                 nvme_ioctl_version,
3821                 nvme_ioctl_format,
3822                 nvme_ioctl_detach,
3823                 nvme_ioctl_attach
3824         };
```

```
3826         if (nvme == NULL)
3827                 return (ENXIO);

3829         if (nsid > nvme->n_namespace_count)
3830                 return (ENXIO);

3832         if (IS_DEVCTL(cmd))
3833                 return (ndi_devctl_ioctl(nvme->n_dip, cmd, arg, mode, 0));

3835 #ifdef _MULTI_DATAMODEL
3836         switch (ddi_model_convert_from(mode & FMODELS)) {
3837         case DDI_MODEL_ILP32: {
3838                 nvme_ioctl32_t nioc32;
3839                 if (ddi_copyin((void*)arg, &nioc32, sizeof (nvme_ioctl32_t),
3840                     mode) != 0)
3841                         return (EFAULT);
3842                 nioc.n_len = nioc32.n_len;
3843                 nioc.n_buf = nioc32.n_buf;
3844                 nioc.n_arg = nioc32.n_arg;
3845                 break;
3846         }
3847         case DDI_MODEL_NONE:
3848 #endif
3849                 if (ddi_copyin((void*)arg, &nioc, sizeof (nvme_ioctl_t), mode)
3850                     != 0)
3851                         return (EFAULT);
3852 #ifdef _MULTI_DATAMODEL
3853                 break;
3854         }
3855 #endif

3857         if (nvme->n_dead && cmd != NVME_IOC_DETACH)
3858                 return (EIO);


3861         if (cmd == NVME_IOC_IDENTIFY_CTRL) {
3862                 /*
3863                  * This makes NVME_IOC_IDENTIFY_CTRL work the same on devctl and
3864                  * attachment point nodes.
3865                  */
3866                 nsid = 0;
3867         } else if (cmd == NVME_IOC_IDENTIFY_NSID && nsid == 0) {
3868                 /*
3869                  * This makes NVME_IOC_IDENTIFY_NSID work on a devctl node, it
3870                  * will always return identify data for namespace 1.
3871                  */
3872                 nsid = 1;
3873         }

3875         if (IS_NVME_IOC(cmd) && nvme_ioctl[NVME_IOC_CMD(cmd)] != NULL)
3876                 rv = nvme_ioctl[NVME_IOC_CMD(cmd)](nvme, nsid, &nioc, mode,
3877                     cred_p);
3878         else
3879                 rv = EINVAL;

3881 #ifdef _MULTI_DATAMODEL
3882         switch (ddi_model_convert_from(mode & FMODELS)) {
3883         case DDI_MODEL_ILP32: {
3884                 nvme_ioctl32_t nioc32;

3886                 nioc32.n_len = (size32_t)nioc.n_len;
3887                 nioc32.n_buf = (uintptr32_t)nioc.n_buf;
3888                 nioc32.n_arg = nioc.n_arg;

3890                 if (ddi_copyout(&nioc32, (void *)arg, sizeof (nvme_ioctl32_t),
```



```
3891                     mode) != 0)
3892                         return (EFAULT);
3893                 break;
3894         }
3895         case DDI_MODEL_NONE:
3896 #endif
3897                 if (ddi_copyout(&nioc, (void *)arg, sizeof (nvme_ioctl_t), mode)
3898                     != 0)
3899                         return (EFAULT);
3900 #ifdef _MULTI_DATAMODEL
3901                 break;
3902         }
3903 #endif

3905         return (rv);
3906 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    5391 Tue Sep 19 12:56:22 2017
new/usr/src/uts/common/io/nvme/nvme_var.h
8629 nvme: rework command abortion
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
Reviewed by: Jason King <jason.king@joyent.com>
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2016 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2016 The MathWorks, Inc. All rights reserved.
 */

#ifndef _NVME_VAR_H
#define _NVME_VAR_H

#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/blkdev.h>
#include <sys/taskq_impl.h>
#include <sys/list.h>

/*
 * NVMe driver state
 */

#ifdef __cplusplus
extern "C" {
#endif

#define NVME_FMA_INIT                   0x1
#define NVME_REGS_MAPPED                0x2
#define NVME_ADMIN_QUEUE                0x4
#define NVME_CTRL_LIMITS                0x8
#define NVME_INTERRUPTS                 0x10

#define NVME_MIN_ADMIN_QUEUE_LEN        16
#define NVME_MIN_IO_QUEUE_LEN           16
#define NVME_DEFAULT_ADMIN_QUEUE_LEN    256
#define NVME_DEFAULT_IO_QUEUE_LEN       1024
#define NVME_DEFAULT_ASYNC_EVENT_LIMIT  10
#define NVME_MIN_ASYNC_EVENT_LIMIT      1
#define NVME_DEFAULT_MIN_BLOCK_SIZE     512


typedef struct nvme nvme_t;
typedef struct nvme_namespace nvme_namespace_t;
typedef struct nvme_minor_state nvme_minor_state_t;
typedef struct nvme_dma nvme_dma_t;
typedef struct nvme_cmd nvme_cmd_t;
typedef struct nvme_qpair nvme_qpair_t;
typedef struct nvme_task_arg nvme_task_arg_t;

struct nvme_minor_state {
        kmutex_t        nm_mutex;
```



```
        boolean_t       nm_oexcl;
        uint_t          nm_ocnt;
};
```

*unchanged_portion_omitted_*

```
struct nvme_cmd {
        struct list_node nc_list;

        nvme_sqe_t nc_sqe;
        nvme_cqe_t nc_cqe;

        void (*nc_callback)(void *);
        bd_xfer_t *nc_xfer;
        boolean_t nc_completed;
        boolean_t nc_dontpanic;
        uint16_t nc_sqid;

        nvme_dma_t *nc_dma;

        kmutex_t nc_mutex;
        kcondvar_t nc_cv;

        taskq_ent_t nc_tqent;
        nvme_t *nc_nvme;
};
```

*unchanged_portion_omitted_*

```
struct nvme {
        dev_info_t *n_dip;
        int n_progress;

        caddr_t n_regs;
        ddi_acc_handle_t n_regh;

        kmem_cache_t *n_cmd_cache;
        kmem_cache_t *n_prp_cache;

        size_t n_inth_sz;
        ddi_intr_handle_t *n_inth;
        int n_intr_cnt;
        uint_t n_intr_pri;
        int n_intr_cap;
        int n_intr_type;
        int n_intr_types;

        char *n_product;
        char *n_vendor;

        nvme_version_t n_version;
        boolean_t n_dead;
        boolean_t n_strict_version;
        boolean_t n_ignore_unknown_vendor_status;
        uint32_t n_admin_queue_len;
        uint32_t n_io_queue_len;
        uint16_t n_async_event_limit;
        uint_t n_min_block_size;
        uint16_t n_abort_command_limit;
        uint64_t n_max_data_transfer_size;
        boolean_t n_write_cache_present;
        boolean_t n_write_cache_enabled;
        int n_error_log_len;
        boolean_t n_lba_range_supported;
        boolean_t n_auto_pst_supported;

        int n_nssr_supported;
        int n_doorbell_stride;
```

```
157         int n_timeout;
158         int n_arbitration_mechanisms;
159         int n_cont_queues_reqd;
160         int n_max_queue_entries;
161         int n_pageshift;
162         int n_pagesize;

164         int n_namespace_count;
165         uint16_t n_ioq_count;
162         int n_ioq_count;

167         nvme_identify_ctrl_t *n_idctl;

169         nvme_qpair_t *n_adminq;
170         nvme_qpair_t **n_ioq;

172         nvme_namespace_t *n_ns;

174         ddi_dma_attr_t n_queue_dma_attr;
175         ddi_dma_attr_t n_prp_dma_attr;
176         ddi_dma_attr_t n_sgl_dma_attr;
177         ddi_device_acc_attr_t n_reg_acc_attr;
178         ddi_iblock_cookie_t n_fm_ibc;
179         int n_fm_cap;

181         ksema_t n_abort_sema;

183         ddi_taskq_t *n_cmd_taskq;

185         /* state for devctl minor node */
186         nvme_minor_state_t n_minor;

188         /* errors detected by driver */
189         uint32_t n_dma_bind_err;
190         uint32_t n_abort_failed;
191         uint32_t n_cmd_timeout;
192         uint32_t n_cmd_aborted;
193         uint32_t n_wrong_logpage;
194         uint32_t n_unknown_logpage;
195         uint32_t n_too_many_cookies;

197         /* errors detected by hardware */
198         uint32_t n_data_xfr_err;
199         uint32_t n_internal_err;
200         uint32_t n_abort_rq_err;
201         uint32_t n_abort_sq_del;
202         uint32_t n_nvm_cap_exc;
203         uint32_t n_nvm_ns_notrdy;
204         uint32_t n_inv_cq_err;
205         uint32_t n_inv_qid_err;
206         uint32_t n_max_qsz_exc;
207         uint32_t n_inv_int_vect;
208         uint32_t n_inv_log_page;
209         uint32_t n_inv_format;
210         uint32_t n_inv_q_del;
211         uint32_t n_cnfl_attr;
212         uint32_t n_inv_prot;
213         uint32_t n_readonly;

215         /* errors reported by asynchronous events */
216         uint32_t n_diagfail_event;
217         uint32_t n_persistent_event;
218         uint32_t n_transient_event;
219         uint32_t n_fw_load_event;
220         uint32_t n_reliability_event;
221         uint32_t n_temperature_event;
```



```
222         uint32_t n_spare_event;
223         uint32_t n_vendor_event;
224         uint32_t n_unknown_event;

226 };
```

*unchanged_portion_omitted_*