

```
*****
10497 Tue Sep 19 12:56:18 2017
new/usr/src/uts/common/io/nvme/nvme.c
don't block in nvme_bd_cmd
8628 nvme: use a semaphore to guard submission queue
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
Reviewed by: Jason King <jason.king@joyent.com>
Reviewed by: Robert Mustacchi <rm@joyent.com>
*****
1 /*
2 * This file and its contents are supplied under the terms of the
3 * Common Development and Distribution License ("CDDL"), version 1.0.
4 * You may only use this file in accordance with the terms of version
5 * 1.0 of the CDDL.
6 *
7 * A full copy of the text of the CDDL should have accompanied this
8 * source. A copy of the CDDL is also available via the Internet at
9 * http://www.illumos.org/license/CDDL.
10 */
11 /*
12 * Copyright 2016 Nexenta Systems, Inc. All rights reserved.
13 * Copyright 2016 Tegile Systems, Inc. All rights reserved.
14 * Copyright (c) 2016 The MathWorks, Inc. All rights reserved.
15 * Copyright 2017 Joyent, Inc.
16 */
17 /*
18 */
19 /*
20 * blkdev driver for NVMe compliant storage devices
21 *
22 * This driver was written to conform to version 1.2.1 of the NVMe
23 * specification. It may work with newer versions, but that is completely
24 * untested and disabled by default.
25 *
26 * The driver has only been tested on x86 systems and will not work on big-
27 * endian systems without changes to the code accessing registers and data
28 * structures used by the hardware.
29 *
30 *
31 * Interrupt Usage:
32 *
33 * The driver will use a single interrupt while configuring the device as the
34 * specification requires, but contrary to the specification it will try to use
35 * a single-message MSI(-X) or FIXED interrupt. Later in the attach process it
36 * will switch to multiple-message MSI(-X) if supported. The driver wants to
37 * have one interrupt vector per CPU, but it will work correctly if less are
38 * available. Interrupts can be shared by queues, the interrupt handler will
39 * iterate through the I/O queue array by steps of n_intr_cnt. Usually only
40 * the admin queue will share an interrupt with one I/O queue. The interrupt
41 * handler will retrieve completed commands from all queues sharing an interrupt
42 * vector and will post them to a taskq for completion processing.
43 *
44 *
45 * Command Processing:
46 *
47 * NVMe devices can have up to 65535 I/O queue pairs, with each queue holding up
48 * NVMe devices can have up to 65536 I/O queue pairs, with each queue holding up
49 * to 65536 I/O commands. The driver will configure one I/O queue pair per
50 * available interrupt vector, with the queue length usually much smaller than
51 * the maximum of 65536. If the hardware doesn't provide enough queues, fewer
52 * interrupt vectors will be used.
53 *
54 * Additionally the hardware provides a single special admin queue pair that can
55 * hold up to 4096 admin commands.
56 *
57 * From the hardware perspective both queues of a queue pair are independent,
```

```
57 * but they share some driver state: the command array (holding pointers to
58 * commands currently being processed by the hardware) and the active command
59 * counter. Access to the submission side of a queue pair and the shared state
60 * is protected by nq_mutex. The completion side of a queue pair does not need
61 * that protection apart from its access to the shared state; it is called only
62 * in the interrupt handler which does not run concurrently for the same
63 * interrupt vector.
64 *
65 * When a command is submitted to a queue pair the active command counter is
66 * incremented and a pointer to the command is stored in the command array. The
67 * array index is used as command identifier (CID) in the submission queue
68 * entry. Some commands may take a very long time to complete, and if the queue
69 * wraps around in that time a submission may find the next array slot to still
70 * be used by a long-running command. In this case the array is sequentially
71 * searched for the next free slot. The length of the command array is the same
72 * as the configured queue length. Queue overrun is prevented by the semaphore,
73 * so a command submission may block if the queue is full.
74 *
75 *
76 * Polled I/O Support:
77 *
78 * For kernel core dump support the driver can do polled I/O. As interrupts are
79 * turned off while dumping the driver will just submit a command in the regular
80 * way, and then repeatedly attempt a command retrieval until it gets the
81 * command back.
82 *
83 *
84 * Namespace Support:
85 *
86 * NVMe devices can have multiple namespaces, each being a independent data
87 * store. The driver supports multiple namespaces and creates a blkdev interface
88 * for each namespace found. Namespaces can have various attributes to support
89 * thin provisioning and protection information. This driver does not support
90 * any of this and ignores namespaces that have these attributes.
91 *
92 * As of NVMe 1.1 namespaces can have an 64bit Extended Unique Identifier
93 * (EUI64). This driver uses the EUI64 if present to generate the devid and
94 * passes it to blkdev to use it in the device node names. As this is currently
95 * untested namespaces with EUI64 are ignored by default.
96 *
97 * We currently support only (2 << NVME_MINOR_INST_SHIFT) - 2 namespaces in a
98 * single controller. This is an artificial limit imposed by the driver to be
99 * able to address a reasonable number of controllers and namespaces using a
100 * 32bit minor node number.
101 *
102 *
103 * Minor nodes:
104 *
105 * For each NVMe device the driver exposes one minor node for the controller and
106 * one minor node for each namespace. The only operations supported by those
107 * minor nodes are open(9E), close(9E), and ioctl(9E). This serves as the
108 * interface for the nvmeadm(1M) utility.
109 *
110 *
111 * Blkdev Interface:
112 *
113 * This driver uses blkdev to do all the heavy lifting involved with presenting
114 * a disk device to the system. As a result, the processing of I/O requests is
115 * relatively simple as blkdev takes care of partitioning, boundary checks, DMA
116 * setup, and splitting of transfers into manageable chunks.
117 *
118 * I/O requests coming in from blkdev are turned into NVM commands and posted to
119 * an I/O queue. The queue is selected by taking the CPU id modulo the number of
120 * queues. There is currently no timeout handling of I/O commands.
```

```

122 * Blkdev also supports querying device/media information and generating a
123 * devid. The driver reports the best block size as determined by the namespace
124 * format back to blkdev as physical block size to support partition and block
125 * alignment. The devid is either based on the namespace EUI64, if present, or
126 * composed using the device vendor ID, model number, serial number, and the
127 * namespace ID.
128 *
129 *
130 * Error Handling:
131 *
132 * Error handling is currently limited to detecting fatal hardware errors,
133 * either by asynchronous events, or synchronously through command status or
134 * admin command timeouts. In case of severe errors the device is fenced off,
135 * all further requests will return EIO. FMA is then called to fault the device.
136 *
137 * The hardware has a limit for outstanding asynchronous event requests. Before
138 * this limit is known the driver assumes it is at least 1 and posts a single
139 * asynchronous request. Later when the limit is known more asynchronous event
140 * requests are posted to allow quicker reception of error information. When an
141 * asynchronous event is posted by the hardware the driver will parse the error
142 * status fields and log information or fault the device, depending on the
143 * severity of the asynchronous event. The asynchronous event request is then
144 * reused and posted to the admin queue again.
145 *
146 * On command completion the command status is checked for errors. In case of
147 * errors indicating a driver bug the driver panics. Almost all other error
148 * status values just cause EIO to be returned.
149 *
150 * Command timeouts are currently detected for all admin commands except
151 * asynchronous event requests. If a command times out and the hardware appears
152 * to be healthy the driver attempts to abort the command. If this fails the
153 * driver assumes the device to be dead, fences it off, and calls FMA to retire
154 * it. In general admin commands are issued at attach time only. No timeout
155 * handling of normal I/O commands is presently done.
156 *
157 * In some cases it may be possible that the ABORT command times out, too. In
158 * that case the device is also declared dead and fenced off.
159 *
160 *
161 * Quiesce / Fast Reboot:
162 *
163 * The driver currently does not support fast reboot. A quiesce(9E) entry point
164 * is still provided which is used to send a shutdown notification to the
165 * device.
166 *
167 *
168 * Driver Configuration:
169 *
170 * The following driver properties can be changed to control some aspects of the
171 * drivers operation:
172 * - strict-version: can be set to 0 to allow devices conforming to newer
173 * versions or namespaces with EUI64 to be used
174 * - ignore-unknown-vendor-status: can be set to 1 to not handle any vendor
175 * specific command status as a fatal error leading device faulting
176 * - admin-queue-len: the maximum length of the admin queue (16-4096)
177 * - io-queue-len: the maximum length of the I/O queues (16-65536)
178 * - async-event-limit: the maximum number of asynchronous event requests to be
179 * posted by the driver
180 * - volatile-write-cache-enable: can be set to 0 to disable the volatile write
181 * cache
182 * - min-phys-block-size: the minimum physical block size to report to blkdev,
183 * which is among other things the basis for ZFS vdev ashift
184 *
185 *
186 * TODO:
187 * - figure out sane default for I/O queue depth reported to blkdev

```

```

188 * - FMA handling of media errors
189 * - support for devices supporting very large I/O requests using chained PRPs
190 * - support for configuring hardware parameters like interrupt coalescing
191 * - support for media formatting and hard partitioning into namespaces
192 * - support for big-endian systems
193 * - support for fast reboot
194 * - support for firmware updates
195 * - support for NVMe Subsystem Reset (1.1)
196 * - support for Scatter/Gather lists (1.1)
197 * - support for Reservations (1.1)
198 * - support for power management
199 */
200 #include <sys/byteorder.h>
201 #ifdef __BIG_ENDIAN
202 #error nvme driver needs porting for big-endian platforms
203 #endif
204
205 #include <sys/modctl.h>
206 #include <sys/conf.h>
207 #include <sys/devops.h>
208 #include <sys/ddi.h>
209 #include <sys/sunddi.h>
210 #include <sys/sundi.h>
211 #include <sys/bitmap.h>
212 #include <sys/sysmacros.h>
213 #include <sys/param.h>
214 #include <sys/varargs.h>
215 #include <sys/cpuvar.h>
216 #include <sys/disp.h>
217 #include <sys/blkdev.h>
218 #include <sys/atomic.h>
219 #include <sys/archsystm.h>
220 #include <sys/sata/sata_hba.h>
221 #include <sys/stat.h>
222 #include <sys/policy.h>
223
224 #include <sys/nvme.h>
225
226 #ifdef __x86
227 #include <sys/x86_archext.h>
228 #endif
229
230
231 #include "nvme_reg.h"
232 #include "nvme_var.h"
233
234 /* NVMe spec version supported */
235 static const int nvme_version_major = 1;
236 static const int nvme_version_minor = 2;
237
238 /* tunable for admin command timeout in seconds, default is 1s */
239 int nvme_admin_cmd_timeout = 1;
240
241 /* tunable for FORMAT NVM command timeout in seconds, default is 600s */
242 int nvme_format_cmd_timeout = 600;
243
244 static int nvme_attach(dev_info_t *, ddi_attach_cmd_t);
245 static int nvme_detach(dev_info_t *, ddi_detach_cmd_t);
246 static int nvme_quiesce(dev_info_t *);
247 static int nvme_fm_errcb(dev_info_t *, ddi_fm_error_t *, const void *);
248 static int nvme_setup_interrupts(nvme_t *, int, int);
249 static void nvme_release_interrupts(nvme_t *);
250 static uint_t nvme_intr(caddr_t, caddr_t);
251
252 static void nvme_shutdown(nvme_t *, int, boolean_t);

```

new/usr/src/uts/common/io/nvme/nvme.c

```
254 static boolean_t nvme_reset(nvme_t *, boolean_t);
255 static int nvme_init(nvme_t *);
256 static nvme_cmd_t *nvme_alloc_cmd(nvme_t *, int);
257 static void nvme_free_cmd(nvme_cmd_t *);
258 static nvme_cmd_t *nvme_create_nvme_cmd(nvme_namespace_t *, uint8_t,
259     bd_xfer_t *);
260 static int nvme_admin_cmd(nvme_cmd_t *, int);
261 static void nvme_submit_admin_cmd(nvme_qpair_t *, nvme_cmd_t *);
262 static int nvme_submit_io_cmd(nvme_qpair_t *, nvme_cmd_t *);
263 static void nvme_submit_cmd_common(nvme_qpair_t *, nvme_cmd_t *);
264 static nvme_cmd_t *nvme_retrieve_cmd(nvme_t *, nvme_qpair_t *);
265 static boolean_t nvme_wait_cmd(nvme_cmd_t *, uint_t);
266 static void nvme_wakeup_cmd(void *);
267 static void nvme_async_event_task(void *);

269 static int nvme_check_unknown_cmd_status(nvme_cmd_t *);
270 static int nvme_check_vendor_cmd_status(nvme_cmd_t *);
271 static int nvme_check_integrity_cmd_status(nvme_cmd_t *);
272 static int nvme_check_specific_cmd_status(nvme_cmd_t *);
273 static int nvme_check_generic_cmd_status(nvme_cmd_t *);
274 static inline int nvme_check_cmd_status(nvme_cmd_t *);

276 static void nvme_abort_cmd(nvme_cmd_t *);
277 static void nvme_async_event(nvme_t *);
278 static int nvme_format_nvme(nvme_t *, uint32_t, uint8_t, boolean_t, uint8_t,
279     boolean_t, uint8_t);
280 static int nvme_get_logpage(nvme_t *, void **, size_t *, uint8_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,
283     uint32_t *);
284 static boolean_t nvme_get_features(nvme_t *, uint32_t, uint8_t, uint32_t *,
285     void **, size_t *);
286 static boolean_t nvme_write_cache_set(nvme_t *, boolean_t);
287 static int nvme_set_nqueues(nvme_t *, uint16_t);

289 static void nvme_free_dma(nvme_dma_t *);
290 static int nvme_zalloc_dma(nvme_t *, size_t, uint_t, ddi_dma_attr_t *,
291     nvme_dma_t **);
292 static int nvme_zalloc_queue_dma(nvme_t *, uint32_t, uint16_t, uint_t,
293     nvme_dma_t **);
294 static void nvme_free_qpair(nvme_qpair_t *);
295 static int nvme_alloc_qpair(nvme_t *, uint32_t, nvme_qpair_t **, int);
296 static int nvme_create_io_qpair(nvme_t *, nvme_qpair_t *, uint16_t);

298 static inline void nvme_put64(nvme_t *, uintptr_t, uint64_t);
299 static inline void nvme_put32(nvme_t *, uintptr_t, uint32_t);
300 static inline uint64_t nvme_get64(nvme_t *, uintptr_t);
301 static inline uint32_t nvme_get32(nvme_t *, uintptr_t);

303 static boolean_t nvme_check_regs_hdl(nvme_t *);
304 static boolean_t nvme_check_dma_hdl(nvme_dma_t *);

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

308 static void nvme_bd_xfer_done(void *);
309 static void nvme_bd_driveinfo(void *, bd_drive_t *);
310 static int nvme_bd_mediainfo(void *, bd_media_t *);
311 static int nvme_bd_cmd(nvme_namespace_t *, bd_xfer_t *, uint8_t);
312 static int nvme_bd_read(void *, bd_xfer_t *);
313 static int nvme_bd_write(void *, bd_xfer_t *);
314 static int nvme_bd_sync(void *, bd_xfer_t *);
315 static int nvme_bd_devid(void *, dev_info_t *, ddi_devid_t *);

317 static int nvme_prp_dma_constructor(void *, void *, int);
```

new/usr/src/uts/common/io/nvme/nvme.

```

318 static void nvme_prp_dma_destructor(void *, void *);
320 static void nvme_prepare_devid(nvme_t *, uint32_t);
322 static int nvme_open(dev_t *, int, int, cred_t *);
323 static int nvme_close(dev_t *, int, int, cred_t *);
324 static int nvme_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

326 #define NVME_MINOR_INST_SHIFT 9
327 #define NVME_MINOR(inst, nsid) (((inst) << NVME_MINOR_INST_SHIFT) | (nsid))
328 #define NVME_MINOR_INST(minor) ((minor) >> NVME_MINOR_INST_SHIFT)
329 #define NVME_MINOR_NSID(minor) ((minor) & ((1 << NVME_MINOR_INST_SHIFT) - 1))
330 #define NVME_MINOR_MAX (NVME_MINOR(1, 0) - 2)

332 static void *nvme_state;
333 static kmem_cache_t *nvme_cmd_cache;

335 /*
336  * DMA attributes for queue DMA memory
337  *
338  * Queue DMA memory must be page aligned. The maximum length of a queue is
339  * 65536 entries, and an entry can be 64 bytes long.
340 */
341 static ddi_dma_attr_t nvme_queue_dma_attr = {
342     .dma_attr_version = DMA_ATTR_V0,
343     .dma_attr_addr_lo = 0,
344     .dma_attr_addr_hi = 0xffffffffffffffffULL,
345     .dma_attr_count_max = (UINT16_MAX + 1) * sizeof (nvme_sqe_t) - 1,
346     .dma_attr_align = 0x1000,
347     .dma_attr_burstsizes = 0x7ff,
348     .dma_attr_minxfer = 0x1000,
349     .dma_attr_maxxfer = (UINT16_MAX + 1) * sizeof (nvme_sqe_t),
350     .dma_attr_seg = 0xffffffffffffffffULL,
351     .dma_attr_sgllen = 1,
352     .dma_attr_granular = 1,
353     .dma_attr_flags = 0,
354 };


---


unchanged portion omitted

721 static void
722 nvme_free_qpair(nvme_qpair_t *qp)
723 {
724     int i;

726     mutex_destroy(&qp->nq_mutex);
727     sema_destroy(&qp->nq_sema);

729     if (qp->nq_sqdma != NULL)
730         nvme_free_dma(qp->nq_sqdma);
731     if (qp->nq_cqdma != NULL)
732         nvme_free_dma(qp->nq_cqdma);

734     if (qp->nq_active_cmds > 0)
735         for (i = 0; i != qp->nq_nentry; i++)
736             if (qp->nq_cmd[i] != NULL)
737                 nvme_free_cmd(qp->nq_cmd[i]);

739     if (qp->nq_cmd != NULL)
740         kmem_free(qp->nq_cmd, sizeof (nvme_cmd_t *) * qp->nq_nentry);

742     kmem_free(qp, sizeof (nvme_qpair_t));
743 }

745 static int
746 nvme_alloc_qpair(nvme_t *nvme, uint32_t nentry, nvme_qpair_t **nqp,
747     int idx)

```

```

748 {
749     nvme_qpair_t *qp = kmem_zalloc(sizeof (*qp), KM_SLEEP);
750
751     mutex_init(&qp->nq_mutex, NULL, MUTEX_DRIVER,
752                DDI_INTR_PRI(nvme->n_intr_pri));
753     sema_init(&qp->nq_sema, nentry, NULL, SEMA_DRIVER, NULL);
754
755     if (nvme_zalloc_queue_dma(nvme, nentry, sizeof (nvme_sqe_t),
756                               DDI_DMA_WRITE, &qp->nq_sqdma) != DDI_SUCCESS)
757         goto fail;
758
759     if (nvme_zalloc_queue_dma(nvme, nentry, sizeof (nvme_cqe_t),
760                               DDI_DMA_READ, &qp->nq_cqdma) != DDI_SUCCESS)
761         goto fail;
762
763     qp->nq_sq = (nvme_sqe_t *)qp->nq_sqdma->nd_memp;
764     qp->nq_cq = (nvme_cqe_t *)qp->nq_cqdma->nd_memp;
765     qp->nq_nentry = nentry;
766
767     qp->nq_sqtdbl = NVME_REG_SQTD dbl(nvme, idx);
768     qp->nq_cqhdbl = NVME_REG_CQHDBL(nvme, idx);
769
770     qp->nq_cmd = kmem_zalloc(sizeof (nvme_cmd_t *) * nentry, KM_SLEEP);
771     qp->nq_next_cmd = 0;
772
773     *nqp = qp;
774     return (DDI_SUCCESS);
775
776 fail:
777     nvme_free_qpair(qp);
778     *nqp = NULL;
779
780     return (DDI_FAILURE);
781 }
unchanged_portion_omitted
820 static void
821 nvme_submit_admin_cmd(nvme_qpair_t *qp, nvme_cmd_t *cmd)
822 {
823     sema_p(&qp->nq_sema);
824     nvme_submit_cmd_common(qp, cmd);
825 }
826
827 static int
828 nvme_submit_io_cmd(nvme_qpair_t *qp, nvme_cmd_t *cmd)
829 {
830     if (sema_try p(&qp->nq_sema) == 0)
831         return (EAGAIN);
832
833     nvme_submit_cmd_common(qp, cmd);
834     return (0);
835 }
836
837 static void
838 nvme_submit_cmd_common(nvme_qpair_t *qp, nvme_cmd_t *cmd)
839 {
840     nvme_reg_sqtdbl_t tail = { 0 };
841
842     mutex_enter(&qp->nq_mutex);
843
844     if (qp->nq_active_cmds == qp->nq_nentry) {
845         mutex_exit(&qp->nq_mutex);
846         return (DDI_FAILURE);
847     }

```

```

843         cmd->nc_completed = B_FALSE;
844
845     /*
846      * Try to insert the cmd into the active cmd array at the nq_next_cmd
847      * slot. If the slot is already occupied advance to the next slot and
848      * try again. This can happen for long running commands like async event
849      * requests.
850      */
851     while (qp->nq_cmd[qp->nq_next_cmd] != NULL)
852         qp->nq_next_cmd = (qp->nq_next_cmd + 1) % qp->nq_nentry;
853     qp->nq_cmd[qp->nq_next_cmd] = cmd;
854
855     qp->nq_active_cmds++;
856
857     cmd->nc_sqe.sqe_cid = qp->nq_next_cmd;
858     bcopy(&cmd->nc_sqe, &qp->nq_sq[qp->nq_sqtail], sizeof (nvme_sqe_t));
859     (void) ddi_dma_sync(qp->nq_sqdma->nd_dmah,
860                         sizeof (nvme_sqe_t) * qp->nq_sqtail,
861                         sizeof (nvme_sqe_t), DDI_DMA_SYNC_FORDEV);
862     qp->nq_next_cmd = (qp->nq_next_cmd + 1) % qp->nq_nentry;
863
864     tail.b.sqtdbl_sqt = qp->nq_sqtail = (qp->nq_sqtail + 1) % qp->nq_nentry;
865     nvme_put32(cmd->nc_nvme, qp->nq_sqtdbl, tail.r);
866
867     mutex_exit(&qp->nq_mutex);
868 }
869
870 static nvme_cmd_t *
871 nvme_retrieve_cmd(nvme_t *nvme, nvme_qpair_t *qp)
872 {
873     nvme_reg_cqhdbl_t head = { 0 };
874
875     nvme_cqe_t *cqe;
876     nvme_cmd_t *cmd;
877
878     (void) ddi_dma_sync(qp->nq_cqdma->nd_dmah, 0,
879                         sizeof (nvme_cqe_t) * qp->nq_nentry, DDI_DMA_SYNC_FORKERNEL);
880
881     mutex_enter(&qp->nq_mutex);
882     cqe = &qp->nq_cq[qp->nq_cqhead];
883
884     /* Check phase tag of CQE. Hardware inverts it for new entries. */
885     if (cqe->cqe_sf.sf_p == qp->nq_phase) {
886         mutex_exit(&qp->nq_mutex);
887         return (NULL);
888     }
889
890     ASSERT(nvme->n_ioq[cqe->cqe_sqid] == qp);
891     ASSERT(cqe->cqe_cid < qp->nq_nentry);
892
893     cmd = qp->nq_cmd[cqe->cqe_cid];
894     qp->nq_cmd[cqe->cqe_cid] = NULL;
895     qp->nq_active_cmds--;
896
897     ASSERT(cmd != NULL);
898     ASSERT(cmd->nc_nvme == nvme);
899     ASSERT(cmd->nc_sqid == cqe->cqe_sqid);
900     ASSERT(cmd->nc_sqe.sqe_cid == cqe->cqe_cid);
901     bcopy(cqe, &cmd->nc_cqe, sizeof (nvme_cqe_t));
902
903     qp->nq_sqhead = cqe->cqe_sqhd;
904
905     head.b.cqhdbl_cqh = qp->nq_cqhead = (qp->nq_cqhead + 1) % qp->nq_nentry;
906
907     /* Toggle phase on wrap-around. */

```

```

908     if (qp->nq_cqhead == 0)
909         qp->nq_phase = qp->nq_phase ? 0 : 1;
910
911     nvme_put32(cmd->nc_nvme, qp->nq_cqhdbl, head.r);
912     mutex_exit(&qp->nq_mutex);
913     sema_v(&qp->nq_sema);
914
915     return (cmd);
916 }
unchanged_portion_omitted
1373 static void
1374 nvme_async_event_task(void *arg)
1375 {
1376     nvme_cmd_t *cmd = arg;
1377     nvme_t *nvme = cmd->nc_nvme;
1378     nvme_error_log_entry_t *error_log = NULL;
1379     nvme_health_log_t *health_log = NULL;
1380     size_t logsize = 0;
1381     nvme_async_event_t event;
1382
1383     /*
1384      * Check for errors associated with the async request itself. The only
1385      * command-specific error is "async event limit exceeded", which
1386      * indicates a programming error in the driver and causes a panic in
1387      * nvme_check_cmd_status().
1388
1389      * Other possible errors are various scenarios where the async request
1390      * was aborted, or internal errors in the device. Internal errors are
1391      * reported to FMA, the command aborts need no special handling here.
1392     */
1393     if (nvme_check_cmd_status(cmd)) {
1394         dev_err(cmd->nc_nvme->n_dip, CE_WARN,
1395                 "!async event request returned failure, sct = %x, "
1396                 "sc = %x, dnr = %d, m = %d", cmd->nc_cqe.cqe_sf.sf_sct,
1397                 cmd->nc_cqe.cqe_sf.sf_sc, cmd->nc_cqe.cqe_sf.sf_dnr,
1398                 cmd->nc_cqe.cqe_sf.m);
1399
1400     if (cmd->nc_cqe.cqe_sf.sf_sct == NVME_CQE_SCT_GENERIC &&
1401         cmd->nc_cqe.cqe_sf.sf_sc == NVME_CQE_SC_GEN_INTERNAL_ERR) {
1402         cmd->nc_nvme->n_dead = B_TRUE;
1403         ddi_fm_service_impact(cmd->nc_nvme->n_dip,
1404                               DDI_SERVICE_LOST);
1405     }
1406     nvme_free_cmd(cmd);
1407     return;
1408 }
1409
1410
1411     event.r = cmd->nc_cqe.cqe_dw0;
1412
1413 /* Clear CQE and re-submit the async request. */
1414 bzero(&cmd->nc_cqe, sizeof (nvme_cqe_t));
1415 nvme_submit_admin_cmd(nvme->n_adming, cmd);
1416 ret = nvme_submit_cmd(nvme->n_adming, cmd);
1417
1418 if (ret != DDI_SUCCESS) {
1419     dev_err(nvme->n_dip, CE_WARN,
1420             "!failed to resubmit async event request");
1421     atomic_inc_32(&nvme->n_async_resubmit_failed);
1422     nvme_free_cmd(cmd);
1423 }
1424
1425 switch (event.b.ae_type) {
1426 case NVME_ASYNC_TYPE_ERROR:

```

```

1427     if (event.b.ae_logpage == NVME_LOGPAGE_ERROR) {
1428         (void) nvme_get_logpage(nvme, (void **) &error_log,
1429                                &logsize, event.b.ae_logpage);
1430     } else {
1431         dev_err(nvme->n_dip, CE_WARN, "!wrong logpage in "
1432                 "async event reply: %d", event.b.ae_logpage);
1433         atomic_inc_32(&nvme->n_wrong_logpage);
1434     }
1435
1436     switch (event.b.ae_info) {
1437     case NVME_ASYNC_ERROR_INV_SQ:
1438         dev_err(nvme->n_dip, CE_PANIC, "programming error: "
1439                 "invalid submission queue");
1440         return;
1441
1442     case NVME_ASYNC_ERROR_INV_DBL:
1443         dev_err(nvme->n_dip, CE_PANIC, "programming error: "
1444                 "invalid doorbell write value");
1445         return;
1446
1447     case NVME_ASYNC_ERROR_DIAGFAIL:
1448         dev_err(nvme->n_dip, CE_WARN, "!diagnostic failure");
1449         ddi_fm_service_impact(nvme->n_dip, DDI_SERVICE_LOST);
1450         nvme->n_dead = B_TRUE;
1451         atomic_inc_32(&nvme->n_diagfail_event);
1452         break;
1453
1454     case NVME_ASYNC_ERROR_PERSISTENT:
1455         dev_err(nvme->n_dip, CE_WARN, "!persistent internal "
1456                 "device error");
1457         ddi_fm_service_impact(nvme->n_dip, DDI_SERVICE_LOST);
1458         nvme->n_dead = B_TRUE;
1459         atomic_inc_32(&nvme->n_persistent_event);
1460         break;
1461
1462     case NVME_ASYNC_ERROR_TRANSIENT:
1463         dev_err(nvme->n_dip, CE_WARN, "!transient internal "
1464                 "device error");
1465         /* TODO: send e report */
1466         atomic_inc_32(&nvme->n_transient_event);
1467         break;
1468
1469     case NVME_ASYNC_ERROR_FW_LOAD:
1470         dev_err(nvme->n_dip, CE_WARN,
1471                 "!firmware image load error");
1472         atomic_inc_32(&nvme->n_fw_load_event);
1473         break;
1474
1475     case NVME_ASYNC_TYPE_HEALTH:
1476         if (event.b.ae_logpage == NVME_LOGPAGE_HEALTH) {
1477             (void) nvme_get_logpage(nvme, (void **) &health_log,
1478                                    &logsize, event.b.ae_logpage, -1);
1479         } else {
1480             dev_err(nvme->n_dip, CE_WARN, "!wrong logpage in "
1481                     "async event reply: %d", event.b.ae_logpage);
1482             atomic_inc_32(&nvme->n_wrong_logpage);
1483         }
1484
1485         switch (event.b.ae_info) {
1486         case NVME_ASYNC_HEALTH_RELIABILITY:
1487             dev_err(nvme->n_dip, CE_WARN,
1488                     "!device reliability compromised");
1489             /* TODO: send e report */
1490             atomic_inc_32(&nvme->n_reliability_event);
1491         }
1492     }
1493 }
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
33
```

```

1485             break;
1486
1487         case NVME_ASYNC_HEALTH_TEMPERATURE:
1488             dev_err(nvme->n_dip, CE_WARN,
1489                     "!temperature above threshold");
1490             /* TODO: send ereport */
1491             atomic_inc_32(&nvme->n_temperature_event);
1492             break;
1493
1494         case NVME_ASYNC_HEALTH_SPARE:
1495             dev_err(nvme->n_dip, CE_WARN,
1496                     "!spare space below threshold");
1497             /* TODO: send ereport */
1498             atomic_inc_32(&nvme->n_spare_event);
1499             break;
1500
1501     }
1502     break;
1503
1504 case NVME_ASYNC_TYPE_VENDOR:
1505     dev_err(nvme->n_dip, CE_WARN, "!vendor specific async event "
1506             "received, info = %x, logpage = %x", event.b.ae_info,
1507             event.b.ae_logpage);
1508     atomic_inc_32(&nvme->n_vendor_event);
1509     break;
1510
1511 default:
1512     dev_err(nvme->n_dip, CE_WARN, "unknown async event received, "
1513             "type = %x, info = %x, logpage = %x", event.b.ae_type,
1514             event.b.ae_info, event.b.ae_logpage);
1515     atomic_inc_32(&nvme->n_unknown_event);
1516     break;
1517
1518 if (error_log)
1519     kmem_free(error_log, logsize);
1520
1521 if (health_log)
1522     kmem_free(health_log, logsize);
1523 }
1524
1525 static int
1526 nvme_admin_cmd(nvme_cmd_t *cmd, int sec)
1527 {
1528     int ret;
1529
1530     mutex_enter(&cmd->nc_mutex);
1531     nvme_submit_admin_cmd(cmd->nc_nvme->n_adminq, cmd);
1532     ret = nvme_submit_cmd(cmd->nc_nvme->n_adminq, cmd);
1533
1534     if (ret != DDI_SUCCESS) {
1535         mutex_exit(&cmd->nc_mutex);
1536         dev_err(cmd->nc_nvme->n_dip, CE_WARN,
1537                 "!nvme_submit_cmd failed");
1538         atomic_inc_32(&cmd->nc_nvme->n_admin_queue_full);
1539         nvme_free_cmd(cmd);
1540         return (DDI_FAILURE);
1541     }
1542
1543     if (nvme_wait_cmd(cmd, sec) == B_FALSE) {
1544         /*
1545          * The command timed out. An abort command was posted that
1546          * will take care of the cleanup.
1547          */
1548         return (DDI_FAILURE);
1549     }
1550     mutex_exit(&cmd->nc_mutex);

```

```

1540         return (DDI_SUCCESS);
1541     }
1542
1543 static void
1544 nvme_async_event(nvme_t *nvme)
1545 {
1546     nvme_cmd_t *cmd = nvme_alloc_cmd(nvme, KM_SLEEP);
1547     int ret;
1548
1549     cmd->nc_sqid = 0;
1550     cmd->nc_sqe.sqe_opc = NVME_OPC_ASYNC_EVENT;
1551     cmd->nc_callback = nvme_async_event_task;
1552
1553     nvme_submit_admin_cmd(nvme->n_adminq, cmd);
1554     ret = nvme_submit_cmd(nvme->n_adminq, cmd);
1555
1556     if (ret != DDI_SUCCESS) {
1557         dev_err(nvme->n_dip, CE_WARN,
1558                 "!nvme_submit_cmd failed for ASYNCHRONOUS EVENT");
1559         nvme_free_cmd(cmd);
1560         return (DDI_FAILURE);
1561     }
1562
1563     return (DDI_SUCCESS);
1564 }
1565
1566 unchanged_portion_omitted
1567
2198 static int
2199 nvme_init(nvme_t *nvme)
2200 {
2201     nvme_reg_cc_t cc = { 0 };
2202     nvme_reg_aqa_t aqa = { 0 };
2203     nvme_reg_asq_t asq = { 0 };
2204     nvme_reg_acq_t acq = { 0 };
2205     nvme_reg_cap_t cap;
2206     nvme_reg_vs_t vs;
2207     nvme_reg_csts_t csts;
2208     int i = 0;
2209     int nqueues;
2210     char model[sizeof(nvme->n_idctl->id_model) + 1];
2211     char *vendor, *product;
2212
2213     /* Check controller version */
2214     vs.r = nvme_get32(nvme, NVME_REG_VS);
2215     nvme->n_version.v_major = vs.b.vs_mjr;
2216     nvme->n_version.v_minor = vs.b.vs_mnr;
2217     dev_err(nvme->n_dip, CE_CONT, "?NVMe spec version %d.%d",
2218             nvme->n_version.v_major, nvme->n_version.v_minor);
2219
2220     if (NVME_VERSION_HIGHER(&nvme->n_version,
2221                             nvme_version_major, nvme_version_minor)) {
2222         dev_err(nvme->n_dip, CE_WARN, "no support for version > %d.%d",
2223                 nvme_version_major, nvme_version_minor);
2224         if (nvme->n_strict_version)
2225             goto fail;
2226     }
2227
2228     /* retrieve controller configuration */
2229     cap.r = nvme_get64(nvme, NVME_REG_CAP);
2230
2231     if ((cap.b.cap_css & NVME_CAP_CSS_NVM) == 0) {
2232         dev_err(nvme->n_dip, CE_WARN,
2233                 "!NVM command set not supported by hardware");
2234         goto fail;
2235     }

```

```

2235     }
2237     nvme->n_nssr_supported = cap.b.cap_nssrs;
2238     nvme->n_doorbell_stride = 4 << cap.b.cap_dstrd;
2239     nvme->n_timeout = cap.b.cap_to;
2240     nvme->n_arbitration_mechanisms = cap.b.cap_ams;
2241     nvme->n_cont_queues_regd = cap.b.cap_cqr;
2242     nvme->n_max_queue_entries = cap.b.cap_mqes + 1;
2243
2244     /*
2245      * The MPSMIN and MPSMAX fields in the CAP register use 0 to specify
2246      * the base page size of 4k (1<<12), so add 12 here to get the real
2247      * page size value.
2248      */
2249     nvme->n_pageshift = MIN(MAX(cap.b.cap_mpsmin + 12, PAGESHIFT),
2250                               cap.b.cap_mpsmax + 12);
2251     nvme->n_pagesize = 1UL << (nvme->n_pageshift);
2252
2253     /*
2254      * Set up Queue DMA to transfer at least 1 page-aligned page at a time.
2255      */
2256     nvme->n_queue_dma_attr.dma_attr_align = nvme->n_pagesize;
2257     nvme->n_queue_dma_attr.dma_attr_minxfer = nvme->n_pagesize;
2258
2259     /*
2260      * Set up PRP DMA to transfer 1 page-aligned page at a time.
2261      * Maxxfer may be increased after we identified the controller limits.
2262      */
2263     nvme->n_prp_dma_attr.dma_attr_maxxfer = nvme->n_pagesize;
2264     nvme->n_prp_dma_attr.dma_attr_minxfer = nvme->n_pagesize;
2265     nvme->n_prp_dma_attr.dma_attr_align = nvme->n_pagesize;
2266     nvme->n_prp_dma_attr.dma_attr_seg = nvme->n_pagesize - 1;
2267
2268     /*
2269      * Reset controller if it's still in ready state.
2270      */
2271     if (nvme_reset(nvme, B_FALSE) == B_FALSE) {
2272         dev_err(nvme->n_dip, CE_WARN, "!unable to reset controller");
2273         ddi_fm_service_impact(nvme->n_dip, DDI_SERVICE_LOST);
2274         nvme->n_dead = B_TRUE;
2275         goto fail;
2276     }
2277
2278     /*
2279      * Create the admin queue pair.
2280      */
2281     if (nvme_alloc_qpair(nvme, nvme->n_admin_queue_len, &nvme->n_admininq, 0)
2282         != DDI_SUCCESS) {
2283         dev_err(nvme->n_dip, CE_WARN,
2284                 "!unable to allocate admin qpair");
2285         goto fail;
2286     }
2287     nvme->n_iocq = kmalloc(sizeof(nvme_qpair_t *), KM_SLEEP);
2288     nvme->n_iocq[0] = nvme->n_admininq;
2289
2290     nvme->n_progress |= NVME_ADMIN_QUEUE;
2291
2292     (void) ddi_prop_update_int(DDI_DEV_T_NONE, nvme->n_dip,
2293                                "admin-queue-len", nvme->n_admin_queue_len);
2294
2295     aqa.b.aqa_asqs = aqa.b.aqa_acqs = nvme->n_admin_queue_len - 1;
2296     asq = nvme->n_admininq->nq_sqdma->nd_cookie.dmac_laddress;
2297     acq = nvme->n_admininq->nq_cqdma->nd_cookie.dmac_laddress;
2298
2299     ASSERT((asq & (nvme->n_pagesize - 1)) == 0);
2300     ASSERT((acq & (nvme->n_pagesize - 1)) == 0);

```

```

2302     nvme_put32(nvme, NVME_REG_AQA, aqa.r);
2303     nvme_put64(nvme, NVME_REG_ASQ, asq);
2304     nvme_put64(nvme, NVME_REG_ACQ, acq);
2305
2306     cc.b.cc_ams = 0; /* use Round-Robin arbitration */
2307     cc.b.cc_css = 0; /* use NVM command set */
2308     cc.b.cc_mps = nvme->n_pageshift - 12;
2309     cc.b.cc_shn = 0; /* no shutdown in progress */
2310     cc.b.cc_en = 1; /* enable controller */
2311     cc.b.cc_iosqes = 6; /* submission queue entry is 2^6 bytes long */
2312     cc.b.cc_iocqes = 4; /* completion queue entry is 2^4 bytes long */
2313
2314     nvme_put32(nvme, NVME_REG_CC, cc.r);
2315
2316     /*
2317      * Wait for the controller to become ready.
2318      */
2319     csts.r = nvme_get32(nvme, NVME_REG_CSTS);
2320     if (csts.b.csts_rdy == 0) {
2321         for (i = 0; i != nvme->n_timeout * 10; i++) {
2322             delay(drv_usectohz(50000));
2323             csts.r = nvme_get32(nvme, NVME_REG_CSTS);
2324
2325             if (csts.b.csts_cfs == 1) {
2326                 dev_err(nvme->n_dip, CE_WARN,
2327                         "!controller fatal status at init");
2328                 ddi_fm_service_impact(nvme->n_dip,
2329                                       DDI_SERVICE_LOST);
2330                 nvme->n_dead = B_TRUE;
2331                 goto fail;
2332             }
2333
2334             if (csts.b.csts_rdy == 1)
2335                 break;
2336         }
2337     }
2338
2339     if (csts.b.csts_rdy == 0) {
2340         dev_err(nvme->n_dip, CE_WARN, "!controller not ready");
2341         ddi_fm_service_impact(nvme->n_dip, DDI_SERVICE_LOST);
2342         nvme->n_dead = B_TRUE;
2343         goto fail;
2344     }
2345
2346     /*
2347      * Assume an abort command limit of 1. We'll destroy and re-init
2348      * that later when we know the true abort command limit.
2349      */
2350     sema_init(&nvme->n_abort_sema, 1, NULL, SEMA_DRIVER, NULL);
2351
2352     /*
2353      * Setup initial interrupt for admin queue.
2354      */
2355     if ((nvme_setup_interrupts(nvme, DDI_INTR_TYPE_MSIX, 1)
2356          != DDI_SUCCESS) ||
2357         (nvme_setup_interrupts(nvme, DDI_INTR_TYPE_MSI, 1)
2358          != DDI_SUCCESS) ||
2359         (nvme_setup_interrupts(nvme, DDI_INTR_TYPE_FIXED, 1)
2360          != DDI_SUCCESS)) {
2361         dev_err(nvme->n_dip, CE_WARN,
2362                 "!failed to setup initial interrupt");
2363         goto fail;
2364     }
2365
2366     /*

```

```

2367     * Post an asynchronous event command to catch errors.
2368     */
2369     nvme_async_event(nvme);
2370     if (nvme_async_event(nvme) != DDI_SUCCESS) {
2371         dev_err(nvme->n_dip, CE_WARN,
2372                 "!failed to post async event");
2373         goto fail;
2374     }
2375
2376     /*
2377      * Identify Controller
2378      */
2379     nvme->n_idctl = nvme_identify(nvme, 0);
2380     if (nvme->n_idctl == NULL) {
2381         dev_err(nvme->n_dip, CE_WARN,
2382                 "!failed to identify controller");
2383         goto fail;
2384     }
2385
2386     /*
2387      * Get Vendor & Product ID
2388      */
2389     bcopy(nvme->n_idctl->id_model, model, sizeof(nvme->n_idctl->id_model));
2390     model[sizeof(nvme->n_idctl->id_model)] = '\0';
2391     sata_split_model(model, &vendor, &product);
2392
2393     if (vendor == NULL)
2394         nvme->n_vendor = strdup("NVMe");
2395     else
2396         nvme->n_vendor = strdup(vendor);
2397
2398     nvme->n_product = strdup(product);
2399
2400     /*
2401      * Get controller limits.
2402      */
2403     nvme->n_async_event_limit = MAX(NVME_MIN_ASYNC_EVENT_LIMIT,
2404                                     MIN(nvme->n_admin_queue_len / 10,
2405                                         MIN(nvme->n_idctl->id_aerl + 1, nvme->n_async_event_limit)));
2406
2407     (void) ddi_prop_update_int(DDI_DEV_T_NONE, nvme->n_dip,
2408                               "async-event-limit", nvme->n_async_event_limit);
2409
2410     nvme->n_abort_command_limit = nvme->n_idctl->id_acl + 1;
2411
2412     /*
2413      * Reinitialize the semaphore with the true abort command limit
2414      * supported by the hardware. It's not necessary to disable interrupts
2415      * as only command aborts use the semaphore, and no commands are
2416      * executed or aborted while we're here.
2417      */
2418     sema_destroy(&nvme->n_abort_sema);
2419     sema_init(&nvme->n_abort_sema, nvme->n_abort_command_limit - 1, NULL,
2420               SEMA_DRIVER, NULL);
2421
2422     nvme->n_progress |= NVME_CTRL_LIMITS;
2423
2424     if (nvme->n_idctl->id_mdts == 0)
2425         nvme->n_max_data_transfer_size = nvme->n_pagesize * 65536;
2426     else
2427         nvme->n_max_data_transfer_size =
2428             lull << (nvme->n_pageshift + nvme->n_idctl->id_mdts);
2429
2430     nvme->n_error_log_len = nvme->n_idctl->id_elpe + 1;
2431
2432     /*

```

```

2433             * Limit n_max_data_transfer_size to what we can handle in one PRP.
2434             * Chained PRPs are currently unsupported.
2435             */
2436             * This is a no-op on hardware which doesn't support a transfer size
2437             * big enough to require chained PRPs.
2438             */
2439             nvme->n_max_data_transfer_size = MIN(nvme->n_max_data_transfer_size,
2440                                                 (nvme->n_pagesize / sizeof(uint64_t) * nvme->n_pagesize));
2441
2442             nvme->n_prp_dma_attr.dma_attr_maxxfer = nvme->n_max_data_transfer_size;
2443
2444             /*
2445              * Make sure the minimum/maximum queue entry sizes are not
2446              * larger/smaller than the default.
2447              */
2448
2449             if (((1 << nvme->n_idctl->id_sqes.qes_min) > sizeof(nvme_sqe_t)) ||
2450                 ((1 << nvme->n_idctl->id_sqes.qes_max) < sizeof(nvme_sqe_t)) ||
2451                 ((1 << nvme->n_idctl->id_cqes.qes_min) > sizeof(nvme_cqe_t)) ||
2452                 ((1 << nvme->n_idctl->id_cqes.qes_max) < sizeof(nvme_cqe_t)))
2453                 goto fail;
2454
2455             /*
2456              * Check for the presence of a Volatile Write Cache. If present,
2457              * enable or disable based on the value of the property
2458              * volatile-write-cache-enable (default is enabled).
2459              */
2460             nvme->n_write_cache_present =
2461                 nvme->n_idctl->id_vwc.vwc_present == 0 ? B_FALSE : B_TRUE;
2462
2463             (void) ddi_prop_update_int(DDI_DEV_T_NONE, nvme->n_dip,
2464                                       "volatile-write-cache-present",
2465                                       nvme->n_write_cache_present ? 1 : 0);
2466
2467             if (!nvme->n_write_cache_present) {
2468                 nvme->n_write_cache_enabled = B_FALSE;
2469             } else if (!nvme->n_write_cache_set(nvme, nvme->n_write_cache_enabled)) {
2470                 dev_err(nvme->n_dip, CE_WARN,
2471                         "!failed to %able volatile write cache",
2472                         nvme->n_write_cache_enabled ? "en" : "dis");
2473                 /*
2474                  * Assume the cache is (still) enabled.
2475                  */
2476                 nvme->n_write_cache_enabled = B_TRUE;
2477             }
2478
2479             (void) ddi_prop_update_int(DDI_DEV_T_NONE, nvme->n_dip,
2480                                       "volatile-write-cache-enable",
2481                                       nvme->n_write_cache_enabled ? 1 : 0);
2482
2483             /*
2484              * Assume LBA Range Type feature is supported. If it isn't this
2485              * will be set to B_FALSE by nvme_get_features().
2486              */
2487             nvme->n_lba_range_supported = B_TRUE;
2488
2489             /*
2490              * Check support for Autonomous Power State Transition.
2491              */
2492             if (NVME_VERSION_ATLEAST(&nvme->n_version, 1, 1))
2493                 nvme->n_auto_pst_supported =
2494                     nvme->n_idctl->id_apsta.ap_sup == 0 ? B_FALSE : B_TRUE;
2495
2496             /*
2497              * Identify Namespaces
2498              */

```

```

2494     nvme->n_namespace_count = nvme->n_idctl->id_nn;
2495     if (nvme->n_namespace_count > NVME_MINOR_MAX) {
2496         dev_err(nvme->n_dip, CE_WARN,
2497                 "!too many namespaces: %d, limiting to %d\n",
2498                 nvme->n_namespace_count, NVME_MINOR_MAX);
2499     nvme->n_namespace_count = NVME_MINOR_MAX;
2500 }
2502     nvme->n_ns = kmalloc(sizeof (nvme_namespace_t) *
2503                           nvme->n_namespace_count, KM_SLEEP);
2505     for (i = 0; i != nvme->n_namespace_count; i++) {
2506         mutex_init(&nvme->n_ns[i].ns_minor.nm_mutex, NULL, MUTEX_DRIVER,
2507                     NULL);
2508         if (nvme_init_ns(nvme, i + 1) != DDI_SUCCESS)
2509             goto fail;
2510 }
2512     /*
2513      * Try to set up MSI/MSI-X interrupts.
2514      */
2515     if ((nvme->n_intr_types & (DDI_INTR_TYPE_MSI | DDI_INTR_TYPE_MSIX))
2516         != 0) {
2516         nvme_release_interrupts(nvme);
2517
2518         nqueues = MIN(UINT16_MAX, ncpus);
2519
2520         if ((nvme_setup_interrupts(nvme, DDI_INTR_TYPE_MSIX,
2521                                   nqueues) != DDI_SUCCESS) ||
2522             (nvme_setup_interrupts(nvme, DDI_INTR_TYPE_MSI,
2523                                   nqueues) != DDI_SUCCESS)) {
2524             dev_err(nvme->n_dip, CE_WARN,
2525                     "!failed to setup MSI/MSI-X interrupts");
2526             goto fail;
2527 }
2528 }
2529
2530     nqueues = nvme->n_intr_cnt;
2531
2532     /*
2533      * Create I/O queue pairs.
2534      */
2535     nvme->n_iog_count = nvme_set_nqueues(nvme, nqueues);
2536     if (nvme->n_iog_count == 0) {
2537         dev_err(nvme->n_dip, CE_WARN,
2538                 "!failed to set number of I/O queues to %d", nqueues);
2539         goto fail;
2540 }
2541
2542     /*
2543      * Reallocate I/O queue array
2544      */
2545     kmem_free(nvme->n_iog, sizeof (nvme_qpair_t *));
2546     nvme->n_iog = kmalloc(sizeof (nvme_qpair_t *) *
2547                           (nvme->n_iog_count + 1), KM_SLEEP);
2548     nvme->n_iog[0] = nvme->n_adminq;
2549
2550     /*
2551      * If we got less queues than we asked for we might as well give
2552      * some of the interrupt vectors back to the system.
2553      */
2554     if (nvme->n_iog_count < nqueues) {
2555         nvme_release_interrupts(nvme);
2556
2557         if (nvme_setup_interrupts(nvme, nvme->n_intr_type,
2558                                   nvme->n_iog_count) != DDI_SUCCESS) {

```

```

2560                         dev_err(nvme->n_dip, CE_WARN,
2561                                 "!failed to reduce number of interrupts");
2562                         goto fail;
2563 }
2564
2565     /*
2566      * Alloc & register I/O queue pairs
2567      */
2568     nvme->n_io_queue_len =
2569             MIN(nvme->n_io_queue_len, nvme->n_max_queue_entries);
2570     (void) ddi_prop_update_int(DDI_DEV_T_NONE, nvme->n_dip, "io-queue-len",
2571                               nvme->n_io_queue_len);
2572
2573     for (i = 1; i != nvme->n_iog_count + 1; i++) {
2574         if (nvme_alloc_qpair(nvme, nvme->n_io_queue_len,
2575                               &nvme->n_iog[i], i) != DDI_SUCCESS) {
2576             dev_err(nvme->n_dip, CE_WARN,
2577                     "!unable to allocate I/O qpair %d", i);
2578             goto fail;
2579 }
2580
2581         if (nvme_create_io_qpair(nvme, nvme->n_iog[i], i)
2582             != DDI_SUCCESS) {
2583             dev_err(nvme->n_dip, CE_WARN,
2584                     "!unable to create I/O qpair %d", i);
2585             goto fail;
2586 }
2587 }
2588
2589     /*
2590      * Post more asynchronous events commands to reduce event reporting
2591      * latency as suggested by the spec.
2592      */
2593     for (i = 1; i != nvme->n_async_event_limit; i++) {
2594         nvme_async_event(nvme);
2595     for (i = 1; i != nvme->n_async_event_limit; i++) {
2596         if (nvme_async_event(nvme) != DDI_SUCCESS) {
2597             dev_err(nvme->n_dip, CE_WARN,
2598                     "!failed to post async event %d", i);
2599             goto fail;
2600 }
2601
2602 } unchanged portion omitted
2603
2604 static int
2605 nvme_bd_cmd(nvme_namespace_t *ns, bd_xfer_t *xfer, uint8_t opc)
2606 {
2607     nvme_t *nvme = ns->ns_nvme;
2608     nvme_cmd_t *cmd;
2609     nvme_cmd_t *cmd, *ret;
2610     nvme_qpair_t *iog;
2611     boolean_t poll;
2612     int ret;
2613
2614     if (nvme->n_dead)
2615         return (EIO);
2616
2617     cmd = nvme_create_nvm_cmd(ns, opc, xfer);

```

```
3268     if (cmd == NULL)
3269         return (ENOMEM);
3271     cmd->nc_sqid = (CPU->cpu_id % nvme->n_ioq_count) + 1;
3272     ASSERT(cmd->nc_sqid <= nvme->n_ioq_count);
3273     ioq = nvme->n_ioq[cmd->nc_sqid];
3275     /*
3276      * Get the polling flag before submitting the command. The command may
3277      * complete immediately after it was submitted, which means we must
3278      * treat both cmd and xfer as if they have been freed already.
3279      */
3280     poll = (xfer->x_flags & BD_XFER_POLL) != 0;
3282     ret = nvme_submit_io_cmd(ioq, cmd);
3283     if (nvme_submit_cmd(ioq, cmd) != DDI_SUCCESS)
3284         return (EAGAIN);
3285     if (ret != 0)
3286         return (ret);
3287     if (!poll)
3288         return (0);
3289     do {
3290         cmd = nvme_retrieve_cmd(nvme, ioq);
3291         if (cmd != NULL)
3292             nvme_bd_xfer_done(cmd);
3293         ret = nvme_retrieve_cmd(nvme, ioq);
3294         if (ret != NULL)
3295             nvme_bd_xfer_done(ret);
3296         else
3297             drv_usecwait(10);
3298     } while (ioq->nq_active_cmds != 0);
3299 }
```

unchanged portion omitted

```
*****
5336 Tue Sep 19 12:56:19 2017
new/usr/src/uts/common/io/nvme/nvme_var.h
8628 nvme: use a semaphore to guard submission queue
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
Reviewed by: Jason King <jason.king@joyent.com>
Reviewed by: Robert Mustacchi <rm@joyent.com>
*****
_____ unchanged_portion_omitted _____
91 struct nvme_qpair {
92     size_t nq_nentry;
94     nvme_dma_t *nq_sqdma;
95     nvme_sqe_t *nq_sq;
96     uint_t nq_sqhead;
97     uint_t nq_sqtail;
98     uintptr_t nq_sqtdbl;
100    nvme_dma_t *nq_cqdma;
101    nvme_cqe_t *nq_cq;
102    uint_t nq_cqhead;
103    uint_t nq_cqtail;
104    uintptr_t nq_cqhdbl;
106    nvme_cmd_t **nq_cmd;
107    uint16_t nq_next_cmd;
108    uint_t nq_active_cmds;
109    int nq_phase;
111    kmutex_t nq_mutex;
112    ksema_t nq_sema;
113 };
115 struct nvme {
116     dev_info_t *n_dip;
117     int n_progress;
119     caddr_t n_regs;
120     ddi_acc_handle_t n_regh;
122     kmem_cache_t *n_cmd_cache;
123     kmem_cache_t *n_prp_cache;
125     size_t n_inth_sz;
126     ddi_intr_handle_t *n_inth;
127     int n_intr_cnt;
128     uint_t n_intr_pri;
129     int n_intr_cap;
130     int n_intr_type;
131     int n_intr_types;
133     char *n_product;
134     char *n_vendor;
136     nvme_version_t n_version;
137     boolean_t n_dead;
138     boolean_t n_strict_version;
139     boolean_t n_ignore_unknown_vendor_status;
140     uint32_t n_admin_queue_len;
141     uint32_t n_io_queue_len;
142     uint16_t n_async_event_limit;
143     uint_t n_min_block_size;
144     uint16_t n_abort_command_limit;
145     uint64_t n_max_data_transfer_size;
146     boolean_t n_write_cache_present;
```

```
147     boolean_t n_write_cache_enabled;
148     int n_error_log_len;
149     boolean_t n_lba_range_supported;
150     boolean_t n_auto_pst_supported;
152     int n_nssr_supported;
153     int n_doorbell_stride;
154     int n_timeout;
155     int n_arbitration_mechanisms;
156     int n_cont_queues_reqd;
157     int n_max_queue_entries;
158     int n_pageshift;
159     int n_pagesize;
161     int n_namespace_count;
162     int n_iocount;
164     nvme_identify_ctrl_t *n_idctl;
166     nvme_qpair_t *n_adming;
167     nvme_qpair_t **n_ioc;
169     nvme_namespace_t *n_ns;
171     ddi_dma_attr_t n_queue_dma_attr;
172     ddi_dma_attr_t n_prp_dma_attr;
173     ddi_dma_attr_t n_sgl_dma_attr;
174     ddi_device_acc_attr_t n_reg_acc_attr;
175     ddi_iblock_cookie_t n_fm_ibc;
176     int n_fm_cap;
178     ksema_t n_abort_sema;
180     ddi_taskq_t *n_cmd_taskq;
182     /* state for devctl minor node */
183     nvme_minor_state_t n_minor;
185     /* errors detected by driver */
186     uint32_t n_dma_bind_err;
187     uint32_t n_abort_failed;
188     uint32_t n_cmd_timeout;
189     uint32_t n_cmd_aborted;
189     uint32_t n_async_resubmit_failed;
190     uint32_t n_wrong_logpage;
191     uint32_t n_unknown_logpage;
192     uint32_t n_too_many_cookies;
193     uint32_t n_admin_queue_full;
194     /* errors detected by hardware */
195     uint32_t n_data_xfr_err;
196     uint32_t n_internal_err;
197     uint32_t n_abort_rq_err;
198     uint32_t n_abort_sq_del;
199     uint32_t n_nvme_cap_exc;
200     uint32_t n_nvme_ns_notrdy;
201     uint32_t n_inv_cq_err;
202     uint32_t n_inv_qid_err;
203     uint32_t n_max_gsz_exc;
204     uint32_t n_inv_int_vect;
205     uint32_t n_inv_log_page;
206     uint32_t n_inv_format;
207     uint32_t n_inv_q_del;
208     uint32_t n_cnfl_attr;
209     uint32_t n_inv_prot;
210     uint32_t n_READONLY;
```

```
212     /* errors reported by asynchronous events */
213     uint32_t n_diagfail_event;
214     uint32_t n_persistent_event;
215     uint32_t n_transient_event;
216     uint32_t n_fw_load_event;
217     uint32_t n_reliability_event;
218     uint32_t n_temperature_event;
219     uint32_t n_spare_event;
220     uint32_t n_vendor_event;
221     uint32_t n_unknown_event;
223 };


---

unchanged portion omitted
```