```
**********************************************************
  104420 Thu Sep  7 16:25:42 2017
new/usr/src/uts/common/io/nvme/nvme.c
8631 only 16 NVMe controllers usable per system due to 18bit minor number limit
Reviewed by: Robert Mustacchi <rm@joyent.com>
Reviewed by: Patrick Mooney <patrick.mooney@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 65536 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 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.
 *
 *
 * 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
```

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



```
 192  * - support for fast reboot
 193  * - support for firmware updates
 194  * - support for NVMe Subsystem Reset (1.1)
 195  * - support for Scatter/Gather lists (1.1)
 196  * - support for Reservations (1.1)
 197  * - support for power management
 198  */

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

 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/sunndi.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>

 224 #include <sys/nvme.h>

 226 #ifdef __x86
 227 #include <sys/x86_archext.h>
 228 #endif

 230 #include "nvme_reg.h"
 231 #include "nvme_var.h"


 234 /* NVMe spec version supported */
 235 static const int nvme_version_major = 1;
 236 static const int nvme_version_minor = 2;

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

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

 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);

 252 static void nvme_shutdown(nvme_t *, int, boolean_t);
 253 static boolean_t nvme_reset(nvme_t *, boolean_t);
 254 static int nvme_init(nvme_t *);
 255 static nvme_cmd_t *nvme_alloc_cmd(nvme_t *, int);
 256 static void nvme_free_cmd(nvme_cmd_t *);
 257 static nvme_cmd_t *nvme_create_nvm_cmd(nvme_namespace_t *, uint8_t,
```

```
258     bd_xfer_t *);
259 static int nvme_admin_cmd(nvme_cmd_t *, int);
260 static int nvme_submit_cmd(nvme_qpair_t *, nvme_cmd_t *);
261 static nvme_cmd_t *nvme_retrieve_cmd(nvme_t *, nvme_qpair_t *);
262 static boolean_t nvme_wait_cmd(nvme_cmd_t *, uint_t);
263 static void nvme_wakeup_cmd(void *);
264 static void nvme_async_event_task(void *);

266 static int nvme_check_unknown_cmd_status(nvme_cmd_t *);
267 static int nvme_check_vendor_cmd_status(nvme_cmd_t *);
268 static int nvme_check_integrity_cmd_status(nvme_cmd_t *);
269 static int nvme_check_specific_cmd_status(nvme_cmd_t *);
270 static int nvme_check_generic_cmd_status(nvme_cmd_t *);
271 static inline int nvme_check_cmd_status(nvme_cmd_t *);

273 static void nvme_abort_cmd(nvme_cmd_t *);
274 static int nvme_async_event(nvme_t *);
275 static int nvme_format_nvm(nvme_t *, uint32_t, uint8_t, boolean_t, uint8_t,
276     boolean_t, uint8_t);
277 static int nvme_get_logpage(nvme_t *, void **, size_t *, uint8_t, ...);
278 static void *nvme_identify(nvme_t *, uint32_t);
279 static boolean_t nvme_set_features(nvme_t *, uint32_t, uint8_t, uint32_t,
280     uint32_t *);
281 static boolean_t nvme_get_features(nvme_t *, uint32_t, uint8_t, uint32_t *,
282     void **, size_t *);
283 static boolean_t nvme_write_cache_set(nvme_t *, boolean_t);
284 static int nvme_set_nqueues(nvme_t *, uint16_t);

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

295 static inline void nvme_put64(nvme_t *, uintptr_t, uint64_t);
296 static inline void nvme_put32(nvme_t *, uintptr_t, uint32_t);
297 static inline uint64_t nvme_get64(nvme_t *, uintptr_t);
298 static inline uint32_t nvme_get32(nvme_t *, uintptr_t);

300 static boolean_t nvme_check_regs_hdl(nvme_t *);
301 static boolean_t nvme_check_dma_hdl(nvme_dma_t *);

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

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

314 static int nvme_prp_dma_constructor(void *, void *, int);
315 static void nvme_prp_dma_destructor(void *, void *);

317 static void nvme_prepare_devid(nvme_t *, uint32_t);

319 static int nvme_open(dev_t *, int, int, cred_t *);
320 static int nvme_close(dev_t, int, int, cred_t *);
321 static int nvme_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

323 #define NVME_MINOR_INST_SHIFT   9
```



```
323 #define NVME_MINOR_INST_SHIFT   14
324 #define NVME_MINOR(inst, nsid)  (((inst) << NVME_MINOR_INST_SHIFT) | (nsid))
325 #define NVME_MINOR_INST(minor)  ((minor) >> NVME_MINOR_INST_SHIFT)
326 #define NVME_MINOR_NSID(minor)  ((minor) & ((1 << NVME_MINOR_INST_SHIFT) - 1))
327 #define NVME_MINOR_MAX          (NVME_MINOR(1, 0) - 2)

329 static void *nvme_state;
330 static kmem_cache_t *nvme_cmd_cache;

332 /*
333  * DMA attributes for queue DMA memory
334  *
335  * Queue DMA memory must be page aligned. The maximum length of a queue is
336  * 65536 entries, and an entry can be 64 bytes long.
337  */
338 static ddi_dma_attr_t nvme_queue_dma_attr = {
339         .dma_attr_version       = DMA_ATTR_V0,
340         .dma_attr_addr_lo       = 0,
341         .dma_attr_addr_hi       = 0xffffffffffffffffULL,
342         .dma_attr_count_max     = (UINT16_MAX + 1) * sizeof (nvme_sqe_t) - 1,
343         .dma_attr_align         = 0x1000,
344         .dma_attr_burstsizes    = 0x7ff,
345         .dma_attr_minxfer       = 0x1000,
346         .dma_attr_maxxfer       = (UINT16_MAX + 1) * sizeof (nvme_sqe_t),
347         .dma_attr_seg           = 0xffffffffffffffffULL,
348         .dma_attr_sgllen        = 1,
349         .dma_attr_granular      = 1,
350         .dma_attr_flags         = 0,
351 };
______unchanged_portion_omitted_
```