```
**********************************************************
   75990 Mon Jul  9 10:24:41 2018
new/usr/src/uts/common/io/usb/hcd/xhci/xhci.c
Add xhci_quiesce to support fast reboot.
**********************************************************
```

```
/*
 * 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 (c) 2017, Joyent, Inc.
 * Copyright (c) 2018, Western Digital Corporation.
 */

/*
 * Extensible Host Controller Interface (xHCI) USB Driver
 *
 * The xhci driver is an HCI driver for USB that bridges the gap between client
 * device drivers and implements the actual way that we talk to devices. The
 * xhci specification provides access to USB 3.x capable devices, as well as all
 * prior generations. Like other host controllers, it both provides the way to
 * talk to devices and also is treated like a hub (often called the root hub).
 *
 * This driver is part of the USBA (USB Architecture). It implements the HCDI
 * (host controller device interface) end of USBA. These entry points are used
 * by the USBA on behalf of client device drivers to access their devices. The
 * driver also provides notifications to deal with hot plug events, which are
 * quite common in USB.
 *
 * ----------------
 * USB Introduction
 * ----------------
 *
 * To properly understand the xhci driver and the design of the USBA HCDI
 * interfaces it implements, it helps to have a bit of background into how USB
 * devices are structured and understand how they work at a high-level.
 *
 * USB devices, like PCI devices, are broken down into different classes of
 * device. For example, with USB you have hubs, human-input devices (keyboards,
 * mice, etc.), mass storage, etc. Every device also has a vendor and device ID.
 * Many client drivers bind to an entire class of device, for example, the hubd
 * driver (to hubs) or scsa2usb (USB storage). However, there are other drivers
 * that bind to explicit IDs such as usbsprl (specific USB to Serial devices).
 *
 * USB SPEEDS AND VERSIONS
 *
 * USB devices are often referred to in two different ways. One way they're
 * described is with the USB version that they conform to. In the wild, you're
 * most likely going to see USB 1.1, 2.0, 2.1, and 3.0. However, you may also
 * see devices referred to as 'full-', 'low-', 'high-', and 'super-' speed
 * devices.
 *
 * The latter description describes the maximum theoretical speed of a given
 * device. For example, a super-speed device theoretically caps out around 5
 * Gbit/s, whereas a low-speed device caps out at 1.5 Mbit/s.
 *
 * In general, each speed usually corresponds to a specific USB protocol
 * generation. For example, all USB 3.0 devices are super-speed devices. All
 * 'high-speed' devices are USB 2.x devices. Full-speed devices are special in
```



```
 * that they can either be USB 1.x or USB 2.x devices. Low-speed devices are
 * only a USB 1.x thing, they did not jump the fire line to USB 2.x.
 *
 * USB 3.0 devices and ports generally have the wiring for both USB 2.0 and USB
 * 3.0. When a USB 3.x device is plugged into a USB 2.0 port or hub, then it
 * will report its version as USB 2.1, to indicate that it is actually a USB 3.x
 * device.
 *
 * USB ENDPOINTS
 *
 * A given USB device is made up of endpoints. A request, or transfer, is made
 * to a specific USB endpoint. These endpoints can provide different services
 * and have different expectations around the size of the data that'll be used
 * in a given request and the periodicity of requests. Endpoints themselves are
 * either used to make one-shot requests, for example, making requests to a mass
 * storage device for a given sector, or for making periodic requests where you
 * end up polling on the endpoint, for example, polling on a USB keyboard for
 * keystrokes.
 *
 * Each endpoint encodes two different pieces of information: a direction and a
 * type. There are two different directions: IN and OUT. These refer to the
 * general direction that data moves relative to the operating system. For
 * example, an IN transfer transfers data in to the operating system, from the
 * device. An OUT transfer transfers data from the operating system, out to the
 * device.
 *
 * There are four different kinds of endpoints:
 *
 *      BULK            These transfers are large transfers of data to or from
 *                      a device. The most common use for bulk transfers is for
 *                      mass storage devices. Though they are often also used by
 *                      network devices and more. Bulk endpoints do not have an
 *                      explicit time component to them. They are always used
 *                      for one-shot transfers.
 *
 *      CONTROL         These transfers are used to manipulate devices
 *                      themselves and are used for USB protocol level
 *                      operations (whether device-specific, class-specific, or
 *                      generic across all of USB). Unlike other transfers,
 *                      control transfers are always bi-directional and use
 *                      different kinds of transfers.
 *
 *      INTERRUPT       Interrupt transfers are used for small transfers that
 *                      happen infrequently, but need reasonable latency. A good
 *                      example of interrupt transfers is to receive input from
 *                      a USB keyboard. Interrupt-IN transfers are generally
 *                      polled. Meaning that a client (device driver) opens up
 *                      an interrupt-IN pipe to poll on it, and receives
 *                      periodic updates whenever there is information
 *                      available. However, Interrupt transfers can be used
 *                      as one-shot transfers both going IN and OUT.
 *
 *      ISOCHRONOUS     These transfers are things that happen once per
 *                      time-interval at a very regular rate. A good example of
 *                      these transfers are for audio and video. A device may
 *                      describe an interval as 10ms at which point it will read
 *                      or write the next batch of data every 10ms and transform
 *                      it for the user. There are no one-shot Isochronous-IN
 *                      transfers. There are one-shot Isochronous-OUT transfers,
 *                      but these are used by device drivers to always provide
 *                      the system with sufficient data.
 *
 * To find out information about the endpoints, USB devices have a series of
 * descriptors that cover different aspects of the device. For example, there
 * are endpoint descriptors which cover the properties of endpoints such as the
 * maximum packet size or polling interval.
```

```
 128  *
 129  * Descriptors exist at all levels of USB. For example, there are general
 130  * descriptors for every device. The USB device descriptor is described in
 131  * usb_dev_descr(9S). Host controllers will look at these descriptors to ensure
 132  * that they program the device correctly; however, they are more often used by
 133  * client device drivers. There are also descriptors that exist at a class
 134  * level. For example, the hub class has a class-specific descriptor which
 135  * describes properties of the hub. That information is requested for and used
 136  * by the hub driver.
 137  *
 138  * All of the different descriptors are gathered by the system and placed into a
 139  * tree which USBA sometimes calls the 'Configuration Cloud'. Client device
 140  * drivers gain access to this cloud and then use them to open endpoints, which
 141  * are called pipes in USBA (and some revisions of the USB specification).
 142  *
 143  * Each pipe gives access to a specific endpoint on the device which can be used
 144  * to perform transfers of a specific type and direction. For example, a mass
 145  * storage device often has three different endpoints, the default control
 146  * endpoint (which every device has), a Bulk-IN endpoint, and a Bulk-OUT
 147  * endpoint. The device driver ends up with three open pipes. One to the default
 148  * control endpoint to configure the device, and then the other two are used to
 149  * perform I/O.
 150  *
 151  * These routines translate more or less directly into calls to a host
 152  * controller driver. A request to open a pipe takes an endpoint descriptor that
 153  * describes the properties of the pipe, and the host controller driver (this
 154  * driver) goes through and does any work necessary to allow the client device
 155  * driver to access it. Once the pipe is open, it either makes one-shot
 156  * transfers specific to the transfer type or it starts performing a periodic
 157  * poll of an endpoint.
 158  *
 159  * All of these different actions translate into requests to the host
 160  * controller. The host controller driver itself is in charge of making sure
 161  * that all of the required resources for polling are allocated with a request
 162  * and then proceed to give the driver's periodic callbacks.
 163  *
 164  * HUBS AND HOST CONTROLLERS
 165  *
 166  * Every device is always plugged into a hub, even if the device is itself a
 167  * hub. This continues until we reach what we call the root-hub. The root-hub is
 168  * special in that it is not an actual USB hub, but is integrated into the host
 169  * controller and is manipulated in its own way. For example, the host
 170  * controller is used to turn on and off a given port's power. This may happen
 171  * over any interface, though the most common way is through PCI.
 172  *
 173  * In addition to the normal character device that exists for a host controller
 174  * driver, as part of attaching, the host controller binds to an instance of the
 175  * hubd driver. While the root-hub is a bit of a fiction, everyone models the
 176  * root-hub as the same as any other hub that's plugged in. The hub kernel
 177  * module doesn't know that the hub isn't a physical device that's been plugged
 178  * in. The host controller driver simulates that view by taking hub requests
 179  * that are made and translating them into corresponding requests that are
 180  * understood by the host controller, for example, reading and writing to a
 181  * memory mapped register.
 182  *
 183  * The hub driver polls for changes in device state using an Interrupt-IN
 184  * request, which is the same as is done for the root-hub. This allows the host
 185  * controller driver to not have to know about the implementation of device hot
 186  * plug, merely react to requests from a hub, the same as if it were an external
 187  * device. When the hub driver detects a change, it will go through the
 188  * corresponding state machine and attach or detach the corresponding client
 189  * device driver, depending if the device was inserted or removed.
 190  *
 191  * We detect the changes for the Interrupt-IN primarily based on the port state
 192  * change events that are delivered to the event ring. Whenever any event is
 193  * fired, we use this to update the hub driver about _all_ ports with
```



```
 194  * outstanding events. This more closely matches how a hub is supposed to behave
 195  * and leaves things less likely for the hub driver to end up without clearing a
 196  * flag on a port.
 197  *
 198  * PACKET SIZES AND BURSTING
 199  *
 200  * A given USB endpoint has an explicit packet size and a number of packets that
 201  * can be sent per time interval. These concepts are abstracted away from client
 202  * device drives usually, though they sometimes inform the upper bounds of what
 203  * a device can perform.
 204  *
 205  * The host controller uses this information to transform arbitrary transfer
 206  * requests into USB protocol packets. One of the nice things about the host
 207  * controllers is that they abstract away all of the signaling and semantics of
 208  * the actual USB protocols, allowing for life to be slightly easier in the
 209  * operating system.
 210  *
 211  * That said, if the host controller is not programmed correctly, these can end
 212  * up causing transaction errors and other problems in response to the data that
 213  * the host controller is trying to send or receive.
 214  *
 215  * ------------
 216  * Organization
 217  * ------------
 218  *
 219  * The driver is made up of the following files. Many of these have their own
 220  * theory statements to describe what they do. Here, we touch on each of the
 221  * purpose of each of these files.
 222  *
 223  * xhci_command.c:      This file contains the logic to issue commands to the
 224  *                      controller as well as the actual functions that the
 225  *                      other parts of the driver use to cause those commands.
 226  *
 227  * xhci_context.c:      This file manages various data structures used by the
 228  *                      controller to manage the controller's and device's
 229  *                      context data structures. See more in the xHCI Overview
 230  *                      and General Design for more information.
 231  *
 232  * xhci_dma.c:          This manages the allocation of DMA memory and DMA
 233  *                      attributes for controller, whether memory is for a
 234  *                      transfer or something else. This file also deals with
 235  *                      all the logic of getting data in and out of DMA buffers.
 236  *
 237  * xhci_endpoint.c:     This manages all of the logic of handling endpoints or
 238  *                      pipes. It deals with endpoint configuration, I/O
 239  *                      scheduling, timeouts, and callbacks to USBA.
 240  *
 241  * xhci_event.c:        This manages callbacks from the hardware to the driver.
 242  *                      This covers command completion notifications and I/O
 243  *                      notifications.
 244  *
 245  * xhci_hub.c:          This manages the virtual root-hub. It basically
 246  *                      implements and translates all of the USB level requests
 247  *                      into xhci specific implements. It also contains the
 248  *                      functions to register this hub with USBA.
 249  *
 250  * xhci_intr.c:         This manages the underlying interrupt allocation,
 251  *                      interrupt moderation, and interrupt routines.
 252  *
 253  * xhci_quirks.c:       This manages information about buggy hardware that's
 254  *                      been collected and experienced primarily from other
 255  *                      systems.
 256  *
 257  * xhci_ring.c:         This manages the abstraction of a ring in xhci, which is
 258  *                      the primary of communication between the driver and the
 259  *                      hardware, whether for the controller or a device.
```

```
 *
 * xhci_usba.c:         This implements all of the HCDI functions required by
 *                      USBA. This is the main entry point that drivers and the
 *                      kernel frameworks will reach to start any operation.
 *                      Many functions here will end up in the command and
 *                      endpoint code.
 *
 * xhci.c:              This provides the main kernel DDI interfaces and
 *                      performs device initialization.
 *
 * xhci.h:              This is the primary header file which defines
 *                      illumos-specific data structures and constants to manage
 *                      the system.
 *
 * xhcireg.h:           This header file defines all of the register offsets,
 *                      masks, and related macros. It also contains all of the
 *                      constants that are used in various structures as defined
 *                      by the specification, such as command offsets, etc.
 *
 * xhci_ioctl.h:        This contains a few private ioctls that are used by a
 *                      private debugging command. These are private.
 *
 * cmd/xhci/xhci_portsc:        This is a private utility that can be useful for
 *                              debugging xhci state. It is the only consumer of
 *                              xhci_ioctl.h and the private ioctls.
 *
 * ----------------------------------
 * xHCI Overview and Structure Layout
 * ----------------------------------
 *
 * The design and structure of this driver follows from the way that the xHCI
 * specification tells us that we have to work with hardware. First we'll give a
 * rough summary of how that works, though the xHCI 1.1 specification should be
 * referenced when going through this.
 *
 * There are three primary parts of the hardware -- registers, contexts, and
 * rings. The registers are memory mapped registers that come in four sets,
 * though all are found within the first BAR. These are used to program and
 * control the hardware and aspects of the devices. Beyond more traditional
 * device programming there are two primary sets of registers that are
 * important:
 *
 *   o Port Status and Control Registers (XHCI_PORTSC)
 *   o Doorbell Array (XHCI_DOORBELL)
 *
 * The port status and control registers are used to get and manipulate the
 * status of a given device. For example, turning on and off the power to it.
 * The Doorbell Array is used to kick off I/O operations and start the
 * processing of an I/O ring.
 *
 * The contexts are data structures that represent various pieces of information
 * in the controller. These contexts are generally filled out by the driver and
 * then acknowledged and consumed by the hardware. There are controller-wide
 * contexts (mostly managed in xhci_context.c) that are used to point to the
 * contexts that exist for each device in the system. The primary context is
 * called the Device Context Base Address Array (DCBAA).
 *
 * Each device in the system is allocated a 'slot', which is used to index into
 * the DCBAA. Slots are assigned based on issuing commands to the controller.
 * There are a fixed number of slots that determine the maximum number of
 * devices that can end up being supported in the system. Note this includes all
 * the devices plugged into the USB device tree, not just devices plugged into
 * ports on the chassis.
 *
 * For each device, there is a context structure that describes properties of
 * the device. For example, what speed is the device, is it a hub, etc. The
```



```
 * context has slots for the device and for each endpoint on the device. As
 * endpoints are enabled, their context information which describes things like
 * the maximum packet size, is filled in and enabled. The mapping between these
 * contexts look like:
 *
 *
 *      DCBAA
 *    +--------+                    Device Context
 *    | Slot 0 |------------------->+--------------+
 *    +--------+                    | Slot Context |
 *    |  ...   |                    +--------------+       +----------+
 *    +--------+   +------+         |  Endpoint 0  |------>| I/O Ring |
 *    | Slot n |-->| NULL |         | Context (Bi) |       +----------+
 *    +--------+   +------+         +--------------+
 *                                  |  Endpoint 1  |
 *                                  | Context (Out)|
 *                                  +--------------+
 *                                  |  Endpoint 1  |
 *                                  | Context (In) |
 *                                  +--------------+
 *                                  |      ...     |
 *                                  +--------------+
 *                                  | Endpoint 15  |
 *                                  | Context (In) |
 *                                  +--------------+
 *
 * These contexts are always owned by the controller, though we can read them
 * after various operations complete. Commands that toggle device state use a
 * specific input context, which is a variant of the device context. The only
 * difference is that it has an input context structure ahead of it to say which
 * sections of the device context should be evaluated.
 *
 * Each active endpoint points us to an I/O ring, which leads us to the third
 * main data structure that's used by the device: rings. Rings are made up of
 * transfer request blocks (TRBs), which are joined together to form a given
 * transfer description (TD) which represents a single I/O request.
 *
 * These rings are used to issue I/O to individual endpoints, to issue commands
 * to the controller, and to receive notification of changes and completions.
 * Issued commands go on the special ring called the command ring while the
 * change and completion notifications go on the event ring.  More details are
 * available in xhci_ring.c. Each of these structures is represented by an
 * xhci_ring_t.
 *
 * Each ring can be made up of one or more disjoint regions of DMA; however, we
 * only use a single one. This also impacts some additional registers and
 * structures that exist. The event ring has an indirection table called the
 * Event Ring Segment Table (ERST). Each entry in the table (a segment)
 * describes a chunk of the event ring.
 *
 * One other thing worth calling out is the scratchpad. The scratchpad is a way
 * for the controller to be given arbitrary memory by the OS that it can use.
 * There are two parts to the scratchpad. The first part is an array whose
 * entries contain pointers to the actual addresses for the pages. The second
 * part that we allocate are the actual pages themselves.
 *
 * ----------------------------
 * Endpoint State and Management
 * ----------------------------
 *
 * Endpoint management is one of the key parts to the xhci driver as every
 * endpoint is a pipe that a device driver uses, so they are our primary
 * currency. Endpoints are enabled and disabled when the client device drivers
 * open and close a pipe. When an endpoint is enabled, we have to fill in an
 * endpoint's context structure with information about the endpoint. These
 * basically tell the controller important properties which it uses to ensure
```

```
392  * that there is adequate bandwidth for the device.
393  *
394  * Each endpoint has its own ring as described in the previous section. We place
395  * TRBs (transfer request blocks) onto a given ring to request I/O be performed.
396  * Responses are placed on the event ring, in other words, the rings associated
397  * with an endpoint are purely for producing I/O.
398  *
399  * Endpoints have a defined state machine as described in xHCI 1.1 / 4.8.3.
400  * These states generally correspond with the state of the endpoint to process
401  * I/O and handle timeouts. The driver basically follows a similar state machine
402  * as described there. There are some deviations. For example, what they
403  * describe as 'running' we break into both the Idle and Running states below.
404  * We also have a notion of timed out and quiescing. The following image
405  * summarizes the states and transitions:
406  *
407  *     +------+                                  +-----------+
408  *     | Idle |---------*---------------------->|  Running  |<-+
409  *     +------+         . I/O queued on          +-----------+  |
410  *        ^               ring and timeout         |  |  |     |
411  *        |               scheduled.               |  |  |     |
412  *        |                                        |  |  |     |
413  *        +-----*----------------------------------+  |  |     |
414  *        |     . No I/Os remain                      |  |     |
415  *        |                                           |  |     |
416  *        |               +------*------------------+  |     |
417  *        |               |      . Timeout             |     |
418  *        |               |        fires for           |     |
419  *        |               |        I/O                 |     |
420  *        |               v                            v     |
421  *        |          +-----------+                  +--------+  |
422  *        |          | Timed Out |                  | Halted |  |
423  *        |          +-----------+                  +--------+  |
424  *        |             |                              |        |
425  *        |             |   +-----------+              |        |
426  *        |             +-->| Quiescing |<-----------+         |
427  *        |                 +-----------+                      |
428  *        |   No TRBs.           |                 . TRBs       |
429  *        |   remain .           |                 . Remain     |
430  *        +----------*----<------+--------->-------*------------+
431  *
432  * Normally, a given endpoint will oscillate between having TRBs scheduled and
433  * not. Every time a new I/O is added to the endpoint, we'll ring the doorbell,
434  * making sure that we're processing the ring, presuming that the endpoint isn't
435  * in one of the error states.
436  *
437  * To detect device hangs, we have an active timeout(9F) per active endpoint
438  * that ticks at a one second rate while we still have TRBs outstanding on an
439  * endpoint. Once all outstanding TRBs have been processed, the timeout will
440  * stop itself and there will be no active checking until the endpoint has I/O
441  * scheduled on it again.
442  *
443  * There are two primary ways that things can go wrong on the endpoint. We can
444  * either have a timeout or an event that transitions the endpoint to the Halted
445  * state. In the halted state, we need to issue explicit commands to reset the
446  * endpoint before removing the I/O.
447  *
448  * The way we handle both a timeout and a halted condition is similar, but the
449  * way they are triggered is different. When we detect a halted condition, we
450  * don't immediately clean it up, and wait for the client device driver (or USBA
451  * on its behalf) to issue a pipe reset. When we detect a timeout, we
452  * immediately take action (assuming no other action is ongoing).
453  *
454  * In both cases, we quiesce the device, which takes care of dealing with taking
455  * the endpoint from whatever state it may be in and taking the appropriate
456  * actions based on the state machine in xHCI 1.1 / 4.8.3. The end of quiescing
457  * leaves the device stopped, which allows us to update the ring's pointer and
```



```
458  * remove any TRBs that are causing problems.
459  *
460  * As part of all this, we ensure that we can only be quiescing the device from
461  * a given path at a time. Any requests to schedule I/O during this time will
462  * generally fail.
463  *
464  * The following image describes the state machine for the timeout logic. It
465  * ties into the image above.
466  *
467  *         +----------+                          +---------+
468  *         | Disabled |-----*------------------->| Enabled |<--+
469  *         +----------+     . TRBs scheduled      +---------+   *. 1 sec timer
470  *             ^              and no active          |  |  |     |  fires and
471  *             |              timer.                 |  |  |     |  another
472  *             |                                     |  |  +--+--+  quiesce, in
473  *             |                                     |  |     |     a bad state,
474  *             +------*------------------------------+  |     ^     or decrement
475  *             |      . 1 sec timer                      |     |     I/O timeout
476  *             |        fires and                        |     |
477  *             |        no TRBs or                       |     +--------------+
478  *             |        endpoint shutdown                |                    |
479  *             |                                         *. . timer counter   |
480  *             ^                                         |    reaches zero    |
481  *             |                                         v                    |
482  *             |                                +--------------+              |
483  *             +-------------*----------------<--| Quiesce ring |->---*-------+
484  *                           . No more           | and fail I/O |     . restart
485  *                             I/Os              +--------------+       timer as
486  *                                                                      more I/Os
487  *
488  * As we described above, when there are active TRBs and I/Os, a 1 second
489  * timeout(9F) will be active. Each second, we decrement a counter on the
490  * current, active I/O until either a new I/O takes the head, or the counter
491  * reaches zero. If the counter reaches zero, then we go through, quiesce the
492  * ring, and then clean things up.
493  *
494  * ------------------
495  * Periodic Endpoints
496  * ------------------
497  *
498  * It's worth calling out periodic endpoints explicitly, as they operate
499  * somewhat differently. Periodic endpoints are limited to Interrupt-IN and
500  * Isochronous-IN. The USBA often uses the term polling for these. That's
501  * because the client only needs to make a single API call; however, they'll
502  * receive multiple callbacks until either an error occurs or polling is
503  * requested to be terminated.
504  *
505  * When we have one of these periodic requests, we end up always rescheduling
506  * I/O requests, as well as, having a specific number of pre-existing I/O
507  * requests to cover the periodic needs, in case of latency spikes. Normally,
508  * when replying to a request, we use the request handle that we were given.
509  * However, when we have a periodic request, we're required to duplicate the
510  * handle before giving them data.
511  *
512  * However, the duplication is a bit tricky. For everything that was duplicated,
513  * the framework expects us to submit data. Because of that we, don't duplicate
514  * them until they are needed. This minimizes the likelihood that we have
515  * outstanding requests to deal with when we encounter a fatal polling failure.
516  *
517  * Most of the polling setup logic happens in xhci_usba.c in
518  * xhci_hcdi_periodic_init(). The consumption and duplication is handled in
519  * xhci_endpoint.c.
520  *
521  * ----------------
522  * Structure Layout
523  * ----------------
```

```
 524  *
 525  * The following images relate the core data structures. The primary structure
 526  * in the system is the xhci_t. This is the per-controller data structure that
 527  * exists for each instance of the driver. From there, each device in the system
 528  * is represented by an xhci_device_t and each endpoint is represented by an
 529  * xhci_endpoint_t. For each client that opens a given endpoint, there is an
 530  * xhci_pipe_t. For each I/O related ring, there is an xhci_ring_t in the
 531  * system.
 532  *
 533  *     +------------------------+
 534  *     | Per-Controller         |
 535  *     | Structure              |
 536  *     | xhci_t                 |
 537  *     |                        |
 538  *     | uint_t              ---+--> Capability regs offset
 539  *     | uint_t              ---+--> Operational regs offset
 540  *     | uint_t              ---+--> Runtime regs offset
 541  *     | uint_t              ---+--> Doorbell regs offset
 542  *     | xhci_state_flags_t  ---+--> Device state flags
 543  *     | xhci_quirks_t       ---+--> Device quirk flags
 544  *     | xhci_capability_t   ---+--> Controller capability structure
 545  *     | xhci_dcbaa_t        ---+-----------------------------------+
 546  *     | xhci_scratchpad_t   ---+---------+                         |
 547  *     | xhci_command_ing_t  ---+------+  |                         v
 548  *     | xhci_event_ring_t   ---+----+ |  |                +---------------------+
 549  *     | xhci_usba_t         ---+--+ | |  |                | Device Context      |
 550  *     +------------------------+  | | |  |                | Base Address        |
 551  *                                 | | |  |                | Array Structure     |
 552  *                                 | | |  |                | xhci_dcbaa_t        |
 553  * +-------------------------------+ | |  |                |                     |
 554  * | +-----------------------------+ |  |  DCBAA KVA <-+--        uint64_t * |
 555  * | |    +----------------------------+  | DMA Buffer <-+-- xhci_dma_buffer_t |
 556  * | |    v                          |    |                +---------------------+
 557  * | | +-------------------------+       +----------------------+
 558  * | | | Event Ring              |                              |
 559  * | | | Management              |                              |
 560  * | | | xhci_event_ring_t       |                              v
 561  * | | |                         |   Event Ring        +----------------------+
 562  * | | | xhci_event_segment_t * --|-> Segment VA        |   Scratchpad (Extra  |
 563  * | | | xhci_dma_buffer_t      --|-> Segment DMA Buf.  |   Controller Memory) |
 564  * | | | xhci_ring_t            --|--+                  |    xhci_scratchpad_t |
 565  * | | +-------------------------+  |      Scratchpad  |                      |
 566  * | |                              | Base Array KVA <-+-          uint64_t * |
 567  * | +------------+                 | Array DMA Buf. <-+-   xhci_dma_buffer_t |
 568  * |              v                 | Scratchpad DMA <-+- xhci_dma_buffer_t * |
 569  * |   +--------------------------+ | Buffer per page  +----------------------+
 570  * |   | Command Ring             | |
 571  * |   | xhci_command_ring_t      | +------------------------------+
 572  * |   |                          |                                |
 573  * |   | xhci_ring_t           --+-> Command Ring --->------------+
 574  * |   | list_t                --+-> Command List                 v
 575  * |   | timeout_id_t          --+-> Timeout State     +----------------------+
 576  * |   | xhci_command_ring_state_t +-> State Flags      | I/O Ring             |
 577  * |   +--------------------------+                    | xhci_ring_t          |
 578  * |                                                   |                      |
 579  * |                                  Ring DMA Buf. <-+-- xhci_dma_buffer_t  |
 580  * |                                    Ring Length <-+--          uint_t   |
 581  * |                                 Ring Entry KVA <-+--      xhci_trb_t * |
 582  * |    +--------------------------+       Ring Head <-+--          uint_t   |
 583  * +--->| USBA State               |       Ring Tail <-+--          uint_t   |
 584  *      | xhci_usba_t              |      Ring Cycle <-+--          uint_t   |
 585  *      |                          |                   +----------------------+
 586  *      | usba_hcdi_ops_t *        -+-> USBA Ops Vector                      ^
 587  *      | usb_dev_dscr_t           -+-> USB Virtual Device Descriptor        |
 588  *      | usb_ss_hub_descr_t       -+-> USB Virtual Hub Descriptor           |
 589  *      | usba_pipe_handle_data_t * +-> Interrupt polling client             |
```



```
 590  *      | usb_intr_req_t          -+-> Interrupt polling request             |
 591  *      | uint32_t                --+-> Interrupt polling device mask        |
 592  *      | list_t                  --+-> Pipe List (Active Users)             |
 593  *      | list_t                  --+------------------+                     |
 594  *      +--------------------------+                  |                     ^
 595  *                                                    |                     |
 596  *                                                    v                     |
 597  *     +-------------------------------+            +---------------+       |
 598  *     | USB Device                     |---------->| USB Device    |--> ... |
 599  *     | xhci_device_t                  |           | xhci_device_t |       |
 600  *     |                                |           +---------------+       |
 601  *     | usb_port_t                  --+-> USB Port plugged into             |
 602  *     | uint8_t                     --+-> Slot Number                       |
 603  *     | boolean_t                   --+-> Address Assigned                  |
 604  *     | usba_device_t *             --+-> USBA Device State                 |
 605  *     | xhci_dma_buffer_t           --+-> Input Context DMA Buffer          |
 606  *     | xhci_input_context_t *      --+-> Input Context KVA                 |
 607  *     | xhci_slot_contex_t *        --+-> Input Slot Context KVA            |
 608  *     | xhci_endpoint_context_t *[] --+-> Input Endpoint Context KVA        |
 609  *     | xhci_dma_buffer_t           --+-> Output Context DMA Buffer         |
 610  *     | xhci_slot_context_t *       --+-> Output Slot Context KVA           ^
 611  *     | xhci_endpoint_context_t *[] --+-> Output Endpoint Context KVA       |
 612  *     | xhci_endpoint_t *[]         --+-> Endpoint Tracking ---+            |
 613  *     +-------------------------------+                        |            |
 614  *                                                              |            |
 615  *                                                              v            |
 616  *     +-------------------------------+          +-----------------+       |
 617  *     | Endpoint Data                  |--------->| Endpoint Data   |--> ... |
 618  *     | xhci_endpoint_t                |          | xhci_endpoint_t |       |
 619  *     |                                |          +-----------------+       |
 620  *     | int                         --+-> Endpoint Number                   |
 621  *     | int                         --+-> Endpoint Type                     |
 622  *     | xhci_endpoint_state_t       --+-> Endpoint State                    |
 623  *     | timeout_id_t                --+-> Endpoint Timeout State            |
 624  *     | usba_pipe_handle_data_t *   --+-> USBA Client Handle                |
 625  *     | xhci_ring_t                 --+-> Endpoint I/O Ring  -------->--------+
 626  *     | list_t                      --+-> Transfer List --------+
 627  *     +-------------------------------+                       |
 628  *                                                             v
 629  *     +-------------------------+                  +--------------------+
 630  *     | Transfer Structure      |----------------->| Transfer Structure |-> ...
 631  *     | xhci_transfer_t         |                  | xhci_transfer_t    |
 632  *     |                         |                  +--------------------+
 633  *     | xhci_dma_buffer_t     --+-> I/O DMA Buffer
 634  *     | uint_t                --+-> Number of TRBs
 635  *     | uint_t                --+-> Short transfer data
 636  *     | uint_t                --+-> Timeout seconds remaining
 637  *     | usb_cr_t              --+-> USB Transfer return value
 638  *     | boolean_t             --+-> Data direction
 639  *     | xhci_trb_t *          --+-> Host-order transfer requests for I/O
 640  *     | usb_isoc_pkt_descr_t * -+-> Isochronous only response data
 641  *     | usb_opaque_t          --+-> USBA Request Handle
 642  *     +-------------------------+
 643  *
 644  * -------------
 645  * Lock Ordering
 646  * -------------
 647  *
 648  * There are three different tiers of locks that exist in the driver. First,
 649  * there is a lock for each controller: xhci_t'xhci_lock. This protects all the
 650  * data for that instance of the controller. If there are multiple instances of
 651  * the xHCI controller in the system, each one is independent and protected
 652  * separately. The two do not share any data.
 653  *
 654  * From there, there are two other, specific locks in the system:
 655  *
```

```
 656  *   o xhci_command_ring_t‘xcr_lock
 657  *   o xhci_device_t‘xd_imtx
 658  *
 659  * There is only one xcr_lock per controller, like the xhci_lock. It protects
 660  * the state of the command ring. However, there is on xd_imtx per device.
 661  * Recall that each device is scoped to a given controller. This protects the
 662  * input slot context for a given device.
 663  *
 664  * There are a few important rules to keep in mind here that are true
 665  * universally throughout the driver:
 666  *
 667  * 1) Always grab the xhci_t‘xhci_lock, before grabbing any of the other locks.
 668  * 2) A given xhci_device_t‘xd_imtx, must be taken before grabbing the
 669  *    xhci_command_ring_t‘xcr_lock.
 670  * 3) A given thread can only hold one of the given xhci_device_t‘xd_imtx locks
 671  *    at a given time. In other words, we should never be manipulating the input
 672  *    context of two different devices at once.
 673  * 4) It is safe to hold the xhci_device_t‘xd_imtx while tearing down the
 674  *    endpoint timer. Conversely, the endpoint specific logic should never enter
 675  *    this lock.
 676  *
 677  * --------------------
 678  * Relationship to EHCI
 679  * --------------------
 680  *
 681  * On some Intel chipsets, a given physical port on the system may be routed to
 682  * one of the EHCI or xHCI controllers. This association can be dynamically
 683  * changed by writing to platform specific registers as handled by the quirk
 684  * logic in xhci_quirk.c.
 685  *
 686  * As these ports may support USB 3.x speeds, we always route all such ports to
 687  * the xHCI controller, when supported. In addition, to minimize disruptions
 688  * from devices being enumerated and attached to the EHCI driver and then
 689  * disappearing, we generally attempt to load the xHCI controller before the
 690  * EHCI controller. This logic is not done in the driver; however, it is done in
 691  * other parts of the kernel like in uts/common/io/consconfig_dacf.c in the
 692  * function consconfig_load_drivres().
 693  *
 694  * -----------
 695  * Future Work
 696  * -----------
 697  *
 698  * The primary future work in this driver spans two different, but related
 699  * areas. The first area is around controller resets and how they tie into FM.
 700  * Presently, we do not have a good way to handle controllers coming and going
 701  * in the broader USB stack or properly reconfigure the device after a reset.
 702  * Secondly, we don’t handle the suspend and resume of devices and drivers.
 703  */

 705 #include <sys/param.h>
 706 #include <sys/modctl.h>
 707 #include <sys/conf.h>
 708 #include <sys/devops.h>
 709 #include <sys/ddi.h>
 710 #include <sys/sunddi.h>
 711 #include <sys/cmn_err.h>
 712 #include <sys/ddifm.h>
 713 #include <sys/pci.h>
 714 #include <sys/class.h>
 715 #include <sys/policy.h>

 717 #include <sys/usb/hcd/xhci/xhci.h>
 718 #include <sys/usb/hcd/xhci/xhci_ioctl.h>

 720 /*
 721  * We want to use the first BAR to access its registers. The regs[] array is
```



```
 722  * ordered based on the rules for the PCI supplement to IEEE 1275. So regs[1]
 723  * will always be the first BAR.
 724  */
 725 #define XHCI_REG_NUMBER 1

 727 /*
 728  * This task queue exists as a global taskq that is used for resetting the
 729  * device in the face of FM or runtime errors. Each instance of the device
 730  * (xhci_t) happens to have a single taskq_dispatch_ent already allocated so we
 731  * know that we should always be able to dispatch such an event.
 732  */
 733 static taskq_t *xhci_taskq;

 735 /*
 736  * Global soft state for per-instance data. Note that we must use the soft state
 737  * routines and cannot use the ddi_set_driver_private() routines. The USB
 738  * framework presumes that it can use the dip’s private data.
 739  */
 740 void *xhci_soft_state;

 742 /*
 743  * This is the time in us that we wait after a controller resets before we
 744  * consider reading any register. There are some controllers that want at least
 745  * 1 ms, therefore we default to 10 ms.
 746  */
 747 clock_t xhci_reset_delay = 10000;

 749 void
 750 xhci_error(xhci_t *xhcip, const char *fmt, ...)
 751 {
 752         va_list ap;

 754         va_start(ap, fmt);
 755         if (xhcip != NULL && xhcip->xhci_dip != NULL) {
 756                 vdev_err(xhcip->xhci_dip, CE_WARN, fmt, ap);
 757         } else {
 758                 vcmn_err(CE_WARN, fmt, ap);
 759         }
 760         va_end(ap);
 761 }
______unchanged_portion_omitted_

1011 int
1012 xhci_check_regs_acc(xhci_t *xhcip)
1013 {
1014         ddi_fm_error_t de;

1016         /*
1017          * Treat cases where we can’t check as fine so we can treat the code
1016          * Treat the case where we can’t check as fine so we can treat the code
1018          * more simply.
1019          */
1020         if (quiesce_active || !DDI_FM_ACC_ERR_CAP(xhcip->xhci_fm_caps))
1019         if (!DDI_FM_ACC_ERR_CAP(xhcip->xhci_fm_caps))
1021                 return (DDI_FM_OK);

1023         ddi_fm_acc_err_get(xhcip->xhci_regs_handle, &de, DDI_FME_VERSION);
1024         ddi_fm_acc_err_clear(xhcip->xhci_regs_handle, DDI_FME_VERSION);
1025         return (de.fme_status);
1026 }
______unchanged_portion_omitted_

1975 /* QUIESCE(9E) to support fast reboot */
1976 int
1977 xhci_quiesce(dev_info_t *dip)
1978 {
```

```
1979         xhci_t *xhcip;

1981         xhcip = ddi_get_soft_state(xhci_soft_state, ddi_get_instance(dip));

1983         return (xhci_controller_stop(xhcip) == 0 &&
1984             xhci_controller_reset(xhcip) == 0 ? DDI_SUCCESS : DDI_FAILURE);
1985 }

1987 static int
1988 xhci_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
1989 {
1990         int ret, inst, route;
1991         xhci_t *xhcip;

1993         if (cmd != DDI_ATTACH)
1994                 return (DDI_FAILURE);

1996         inst = ddi_get_instance(dip);
1997         if (ddi_soft_state_zalloc(xhci_soft_state, inst) != 0)
1998                 return (DDI_FAILURE);
1999         xhcip = ddi_get_soft_state(xhci_soft_state, ddi_get_instance(dip));
2000         xhcip->xhci_dip = dip;

2002         xhcip->xhci_regs_capoff = PCI_EINVAL32;
2003         xhcip->xhci_regs_operoff = PCI_EINVAL32;
2004         xhcip->xhci_regs_runoff = PCI_EINVAL32;
2005         xhcip->xhci_regs_dooroff = PCI_EINVAL32;

2007         xhci_fm_init(xhcip);
2008         xhcip->xhci_seq |= XHCI_ATTACH_FM;

2010         if (pci_config_setup(xhcip->xhci_dip, &xhcip->xhci_cfg_handle) !=
2011             DDI_SUCCESS) {
2012                 goto err;
2013         }
2014         xhcip->xhci_seq |= XHCI_ATTACH_PCI_CONFIG;
2015         xhcip->xhci_vendor_id = pci_config_get16(xhcip->xhci_cfg_handle,
2016             PCI_CONF_VENID);
2017         xhcip->xhci_device_id = pci_config_get16(xhcip->xhci_cfg_handle,
2018             PCI_CONF_DEVID);

2020         if (xhci_regs_map(xhcip) == B_FALSE) {
2021                 goto err;
2022         }

2024         xhcip->xhci_seq |= XHCI_ATTACH_REGS_MAP;

2026         if (xhci_regs_init(xhcip) == B_FALSE)
2027                 goto err;

2029         if (xhci_read_params(xhcip) == B_FALSE)
2030                 goto err;

2032         if (xhci_identify(xhcip) == B_FALSE)
2033                 goto err;

2035         if (xhci_alloc_intrs(xhcip) == B_FALSE)
2036                 goto err;
2037         xhcip->xhci_seq |= XHCI_ATTACH_INTR_ALLOC;

2039         if (xhci_add_intr_handler(xhcip) == B_FALSE)
2040                 goto err;
2041         xhcip->xhci_seq |= XHCI_ATTACH_INTR_ADD;

2043         mutex_init(&xhcip->xhci_lock, NULL, MUTEX_DRIVER,
2044             (void *)(uintptr_t)xhcip->xhci_intr_pri);
```



```
2045         cv_init(&xhcip->xhci_statecv, NULL, CV_DRIVER, NULL);
2046         xhcip->xhci_seq |= XHCI_ATTACH_SYNCH;

2048         if (xhci_port_count(xhcip) == B_FALSE)
2049                 goto err;

2051         if (xhci_controller_takeover(xhcip) == B_FALSE)
2052                 goto err;

2054         /*
2055          * We don't enable interrupts until after we take over the controller
2056          * from the BIOS. We've observed cases where this can cause spurious
2057          * interrupts.
2058          */
2059         if (xhci_ddi_intr_enable(xhcip) == B_FALSE)
2060                 goto err;
2061         xhcip->xhci_seq |= XHCI_ATTACH_INTR_ENABLE;

2063         if ((ret = xhci_controller_stop(xhcip)) != 0) {
2064                 xhci_error(xhcip, "failed to stop controller: %s",
2065                     ret == EIO ? "encountered FM register error" :
2066                     "timed out while waiting for controller");
2067                 goto err;
2068         }

2070         if ((ret = xhci_controller_reset(xhcip)) != 0) {
2071                 xhci_error(xhcip, "failed to reset controller: %s",
2072                     ret == EIO ? "encountered FM register error" :
2073                     "timed out while waiting for controller");
2074                 goto err;
2075         }

2077         if ((ret = xhci_controller_configure(xhcip)) != 0) {
2078                 xhci_error(xhcip, "failed to configure controller: %d", ret);
2079                 goto err;
2080         }

2082         /*
2083          * Some systems support having ports routed to both an ehci and xhci
2084          * controller. If we support it and the user hasn't requested otherwise
2085          * via a driver.conf tuning, we reroute it now.
2086          */
2087         route = ddi_prop_get_int(DDI_DEV_T_ANY, xhcip->xhci_dip,
2088             DDI_PROP_DONTPASS, "xhci-reroute", XHCI_PROP_REROUTE_DEFAULT);
2089         if (route != XHCI_PROP_REROUTE_DISABLE &&
2090             (xhcip->xhci_quirks & XHCI_QUIRK_INTC_EHCI))
2091                 (void) xhci_reroute_intel(xhcip);

2093         if ((ret = xhci_controller_start(xhcip)) != 0) {
2094                 xhci_log(xhcip, "failed to reset controller: %s",
2095                     ret == EIO ? "encountered FM register error" :
2096                     "timed out while waiting for controller");
2097                 goto err;
2098         }
2099         xhcip->xhci_seq |= XHCI_ATTACH_STARTED;

2101         /*
2102          * Finally, register ourselves with the USB framework itself.
2103          */
2104         if ((ret = xhci_hcd_init(xhcip)) != 0) {
2105                 xhci_error(xhcip, "failed to register hcd with usba");
2106                 goto err;
2107         }
2108         xhcip->xhci_seq |= XHCI_ATTACH_USBA;

2110         if ((ret = xhci_root_hub_init(xhcip)) != 0) {
```

```
2111                xhci_error(xhcip, "failed to load the root hub driver");
2112                goto err;
2113        }
2114        xhcip->xhci_seq |= XHCI_ATTACH_ROOT_HUB;

2116        return (DDI_SUCCESS);

2118 err:
2119        (void) xhci_cleanup(xhcip);
2120        return (DDI_FAILURE);
2121 }
```

unchanged_portion_omitted_

```
2187 static struct dev_ops xhci_dev_ops = {
2188        DEVO_REV,                       /* devo_rev */
2189        0,                              /* devo_refcnt */
2190        xhci_getinfo,                   /* devo_getinfo */
2191        nulldev,                        /* devo_identify */
2192        nulldev,                        /* devo_probe */
2193        xhci_attach,                    /* devo_attach */
2194        xhci_detach,                    /* devo_detach */
2195        nodev,                          /* devo_reset */
2196        &xhci_cb_ops,                   /* devo_cb_ops */
2197        &usba_hubdi_busops,             /* devo_bus_ops */
2198        usba_hubdi_root_hub_power,      /* devo_power */
2199        xhci_quiesce                    /* devo_quiesce */
2186        ddi_quiesce_not_supported       /* devo_quiesce */
2200 };
```

unchanged_portion_omitted_