```
**********************************************************
  235818 Mon May  5 11:11:19 2014
new/usr/src/uts/common/io/usb/usba/hubdi.c
4782 usba shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1998, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2012 Garrett D'Amore <garrett@damore.org>.  All rights reserved.
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

/*
 * USBA: Solaris USB Architecture support for the hub
 * including root hub
 * Most of the code for hubd resides in this file and
 * is shared between the HCD root hub support and hubd
 */
#define USBA_FRAMEWORK
#include <sys/usb/usba.h>
#include <sys/usb/usba/usba_devdb.h>
#include <sys/sunndi.h>
#include <sys/usb/usba/usba_impl.h>
#include <sys/usb/usba/usba_types.h>
#include <sys/usb/usba/hubdi.h>
#include <sys/usb/usba/hcdi_impl.h>
#include <sys/usb/hubd/hub.h>
#include <sys/usb/hubd/hubdvar.h>
#include <sys/usb/hubd/hubd_impl.h>
#include <sys/kobj.h>
#include <sys/kobj_lex.h>
#include <sys/fs/dv_node.h>
#include <sys/strsun.h>

/*
 * External functions
 */
extern boolean_t consconfig_console_is_ready(void);

/*
 * Prototypes for static functions
 */
static  int     usba_hubdi_bus_ctl(
                        dev_info_t              *dip,
                        dev_info_t              *rdip,
```



```
                        ddi_ctl_enum_t          op,
                        void                    *arg,
                        void                    *result);

static int      usba_hubdi_map_fault(
                        dev_info_t              *dip,
                        dev_info_t              *rdip,
                        struct hat              *hat,
                        struct seg              *seg,
                        caddr_t                 addr,
                        struct devpage          *dp,
                        pfn_t                   pfn,
                        uint_t                  prot,
                        uint_t                  lock);

static int hubd_busop_get_eventcookie(dev_info_t *dip,
                        dev_info_t *rdip,
                        char *eventname,
                        ddi_eventcookie_t *cookie);
static int hubd_busop_add_eventcall(dev_info_t *dip,
                        dev_info_t *rdip,
                        ddi_eventcookie_t cookie,
                        void (*callback)(dev_info_t *dip,
                                ddi_eventcookie_t cookie, void *arg,
                                void *bus_impldata),
                        void *arg, ddi_callback_id_t *cb_id);
static int hubd_busop_remove_eventcall(dev_info_t *dip,
                        ddi_callback_id_t cb_id);
static int hubd_bus_config(dev_info_t *dip,
                        uint_t flag,
                        ddi_bus_config_op_t op,
                        void *arg,
                        dev_info_t **child);
static int hubd_bus_unconfig(dev_info_t *dip,
                        uint_t flag,
                        ddi_bus_config_op_t op,
                        void *arg);
static int hubd_bus_power(dev_info_t *dip, void *impl_arg,
                        pm_bus_power_op_t op, void *arg, void *result);

static usb_port_t  hubd_get_port_num(hubd_t *, struct devctl_iocdata *);
static dev_info_t *hubd_get_child_dip(hubd_t *, usb_port_t);
static uint_t hubd_cfgadm_state(hubd_t *, usb_port_t);
static int hubd_toggle_port(hubd_t *, usb_port_t);
static void hubd_register_cpr_callback(hubd_t *);
static void hubd_unregister_cpr_callback(hubd_t *);

/*
 * Busops vector for USB HUB's
 */
struct bus_ops usba_hubdi_busops =      {
        BUSO_REV,
        nullbusmap,                     /* bus_map */
        NULL,                           /* bus_get_intrspec */
        NULL,                           /* bus_add_intrspec */
        NULL,                           /* bus_remove_intrspec */
        usba_hubdi_map_fault,           /* bus_map_fault */
        NULL,                           /* bus_dma_map */
        ddi_dma_allochdl,
        ddi_dma_freehdl,
        ddi_dma_bindhdl,
        ddi_dma_unbindhdl,
        ddi_dma_flush,
        ddi_dma_win,
        ddi_dma_mctl,                   /* bus_dma_ctl */
        usba_hubdi_bus_ctl,             /* bus_ctl */
```

```
127         ddi_bus_prop_op,                /* bus_prop_op */
128         hubd_busop_get_eventcookie,
129         hubd_busop_add_eventcall,
130         hubd_busop_remove_eventcall,
131         NULL,                           /* bus_post_event */
132         NULL,                           /* bus_intr_ctl */
133         hubd_bus_config,                /* bus_config */
134         hubd_bus_unconfig,              /* bus_unconfig */
135         NULL,                           /* bus_fm_init */
136         NULL,                           /* bus_fm_fini */
137         NULL,                           /* bus_fm_access_enter */
138         NULL,                           /* bus_fm_access_exit */
139         hubd_bus_power                  /* bus_power */
140 };

142 #define USB_HUB_INTEL_VID       0x8087
143 #define USB_HUB_INTEL_PID       0x0020

145 /*
146  * local variables
147  */
148 static kmutex_t usba_hubdi_mutex;       /* protects USBA HUB data structures */

150 static usba_list_entry_t        usba_hubdi_list;

152 usb_log_handle_t        hubdi_log_handle;
153 uint_t                  hubdi_errlevel = USB_LOG_L4;
154 uint_t                  hubdi_errmask = (uint_t)-1;
155 uint8_t                 hubdi_min_pm_threshold = 5; /* seconds */
156 uint8_t                 hubdi_reset_delay = 20; /* seconds */
157 extern int modrootloaded;

159 /*
160  * initialize private data
161  */
162 void
163 usba_hubdi_initialization()
164 {
165         hubdi_log_handle = usb_alloc_log_hdl(NULL, "hubdi", &hubdi_errlevel,
166             &hubdi_errmask, NULL, 0);

168         USB_DPRINTF_L4(DPRINT_MASK_HUBDI, hubdi_log_handle,
169             "usba_hubdi_initialization");

171         mutex_init(&usba_hubdi_mutex, NULL, MUTEX_DRIVER, NULL);

173         usba_init_list(&usba_hubdi_list, NULL, NULL);
174 }


177 void
178 usba_hubdi_destroy()
179 {
180         USB_DPRINTF_L4(DPRINT_MASK_HUBDI, hubdi_log_handle,
181             "usba_hubdi_destroy");

183         mutex_destroy(&usba_hubdi_mutex);
184         usba_destroy_list(&usba_hubdi_list);

186         usb_free_log_hdl(hubdi_log_handle);
187 }


190 /*
191  * Called by an HUB to attach an instance of the driver
192  *      make this instance known to USBA
```



```
193  *      the HUB should initialize usba_hubdi structure prior
194  *      to calling this interface
195  */
196 int
197 usba_hubdi_register(dev_info_t  *dip,
198                 uint_t          flags)
199 {
200         usba_hubdi_t *hubdi = kmem_zalloc(sizeof (usba_hubdi_t), KM_SLEEP);
201         usba_device_t *usba_device = usba_get_usba_device(dip);

203         USB_DPRINTF_L4(DPRINT_MASK_HUBDI, hubdi_log_handle,
204             "usba_hubdi_register: %s", ddi_node_name(dip));

206         hubdi->hubdi_dip = dip;
207         hubdi->hubdi_flags = flags;

209         usba_device->usb_hubdi = hubdi;

211         /*
212          * add this hubdi instance to the list of known hubdi's
213          */
214         usba_init_list(&hubdi->hubdi_list, (usb_opaque_t)hubdi,
215             usba_hcdi_get_hcdi(usba_device->usb_root_hub_dip)->
216             hcdi_iblock_cookie);
217         mutex_enter(&usba_hubdi_mutex);
218         usba_add_to_list(&usba_hubdi_list, &hubdi->hubdi_list);
219         mutex_exit(&usba_hubdi_mutex);

221         return (DDI_SUCCESS);
222 }


225 /*
226  * Called by an HUB to detach an instance of the driver
227  */
228 int
229 usba_hubdi_unregister(dev_info_t *dip)
230 {
231         usba_device_t *usba_device = usba_get_usba_device(dip);
232         usba_hubdi_t *hubdi = usba_device->usb_hubdi;

234         USB_DPRINTF_L4(DPRINT_MASK_HUBDI, hubdi_log_handle,
235             "usba_hubdi_unregister: %s", ddi_node_name(dip));

237         mutex_enter(&usba_hubdi_mutex);
238         (void) usba_rm_from_list(&usba_hubdi_list, &hubdi->hubdi_list);
239         mutex_exit(&usba_hubdi_mutex);

241         usba_destroy_list(&hubdi->hubdi_list);

243         kmem_free(hubdi, sizeof (usba_hubdi_t));

245         return (DDI_SUCCESS);
246 }


249 /*
250  * misc bus routines currently not used
251  */
252 /*ARGSUSED*/
253 static int
254 usba_hubdi_map_fault(dev_info_t *dip,
255         dev_info_t      *rdip,
256         struct hat      *hat,
257         struct seg      *seg,
258         caddr_t         addr,
```

```
259         struct devpage  *dp,
260         pfn_t           pfn,
261         uint_t          prot,
262         uint_t          lock)
263 {
264         return (DDI_FAILURE);
265 }


268 /*
269  * root hub support. the root hub uses the same devi as the HCD
270  */
271 int
272 usba_hubdi_bind_root_hub(dev_info_t *dip,
273         uchar_t *root_hub_config_descriptor,
274         size_t config_length,
275         usb_dev_descr_t *root_hub_device_descriptor)
276 {
277         usba_device_t *usba_device;
278         usba_hcdi_t *hcdi = usba_hcdi_get_hcdi(dip);
279         hubd_t  *root_hubd;
280         usb_pipe_handle_t ph = NULL;
281         dev_info_t *child = ddi_get_child(dip);

283         if (ndi_prop_create_boolean(DDI_DEV_T_NONE, dip,
284             "root-hub") != NDI_SUCCESS) {

286                 return (USB_FAILURE);
287         }

289         usba_add_root_hub(dip);

291         root_hubd = kmem_zalloc(sizeof (hubd_t), KM_SLEEP);

293         /*
294          * create and initialize a usba_device structure
295          */
296         usba_device = usba_alloc_usba_device(dip);

298         mutex_enter(&usba_device->usb_mutex);
299         usba_device->usb_hcdi_ops = hcdi->hcdi_ops;
300         usba_device->usb_cfg = root_hub_config_descriptor;
301         usba_device->usb_cfg_length = config_length;
302         usba_device->usb_dev_descr = root_hub_device_descriptor;
303         usba_device->usb_port = 1;
304         usba_device->usb_addr = ROOT_HUB_ADDR;
305         usba_device->usb_root_hubd = root_hubd;
306         usba_device->usb_cfg_array = kmem_zalloc(sizeof (uchar_t *),
307             KM_SLEEP);
308         usba_device->usb_cfg_array_length = sizeof (uchar_t *);

310         usba_device->usb_cfg_array_len = kmem_zalloc(sizeof (uint16_t),
311             KM_SLEEP);
312         usba_device->usb_cfg_array_len_length = sizeof (uint16_t);

314         usba_device->usb_cfg_array[0] = root_hub_config_descriptor;
315         usba_device->usb_cfg_array_len[0] =
316             sizeof (root_hub_config_descriptor);

318         usba_device->usb_cfg_str_descr = kmem_zalloc(sizeof (uchar_t *),
319             KM_SLEEP);
320         usba_device->usb_n_cfgs = 1;
321         usba_device->usb_n_ifs = 1;
322         usba_device->usb_dip = dip;

324         usba_device->usb_client_flags = kmem_zalloc(
```



```
325             usba_device->usb_n_ifs * USBA_CLIENT_FLAG_SIZE, KM_SLEEP);

327         usba_device->usb_client_attach_list = kmem_zalloc(
328             usba_device->usb_n_ifs *
329             sizeof (*usba_device->usb_client_attach_list), KM_SLEEP);

331         usba_device->usb_client_ev_cb_list = kmem_zalloc(
332             usba_device->usb_n_ifs *
333             sizeof (*usba_device->usb_client_ev_cb_list), KM_SLEEP);

335         /*
336          * The bDeviceProtocol field of root hub device specifies,
337          * whether root hub is a High or Full speed usb device.
338          */
339         if (root_hub_device_descriptor->bDeviceProtocol) {
340                 usba_device->usb_port_status = USBA_HIGH_SPEED_DEV;
341         } else {
342                 usba_device->usb_port_status = USBA_FULL_SPEED_DEV;
343         }

345         mutex_exit(&usba_device->usb_mutex);

347         usba_set_usba_device(dip, usba_device);

349         /*
350          * For the root hub the default pipe is not yet open
351          */
352         if (usb_pipe_open(dip, NULL, NULL,
353             USB_FLAGS_SLEEP | USBA_FLAGS_PRIVILEGED, &ph) != USB_SUCCESS) {
354                 goto fail;
355         }

357         /*
358          * kill off all OBP children, they may not be fully
359          * enumerated
360          */
361         while (child) {
362                 dev_info_t *next = ddi_get_next_sibling(child);
363                 (void) ddi_remove_child(child, 0);
364                 child = next;
365         }

367         /*
368          * "attach" the root hub driver
369          */
370         if (usba_hubdi_attach(dip, DDI_ATTACH) != DDI_SUCCESS) {
371                 goto fail;
372         }

374         return (USB_SUCCESS);

376 fail:
377         (void) ndi_prop_remove(DDI_DEV_T_NONE, dip, "root-hub");

379         usba_rem_root_hub(dip);

381         if (ph) {
382                 usb_pipe_close(dip, ph,
383                     USB_FLAGS_SLEEP | USBA_FLAGS_PRIVILEGED, NULL, NULL);
384         }

386         kmem_free(usba_device->usb_cfg_array,
387             usba_device->usb_cfg_array_length);
388         kmem_free(usba_device->usb_cfg_array_len,
389             usba_device->usb_cfg_array_len_length);
```

```
391         kmem_free(usba_device->usb_cfg_str_descr, sizeof (uchar_t *));

393         usba_free_usba_device(usba_device);

395         usba_set_usba_device(dip, NULL);
396         if (root_hubd) {
397                 kmem_free(root_hubd, sizeof (hubd_t));
398         }

400         return (USB_FAILURE);
401 }


404 int
405 usba_hubdi_unbind_root_hub(dev_info_t *dip)
406 {
407         usba_device_t *usba_device;

409         /* was root hub attached? */
410         if (!(usba_is_root_hub(dip))) {

412                 /* return success anyway */
413                 return (USB_SUCCESS);
414         }

416         /*
417          * usba_hubdi_detach also closes the default pipe
418          * and removes properties so there is no need to
419          * do it here
420          */
421         if (usba_hubdi_detach(dip, DDI_DETACH) != DDI_SUCCESS) {

423                 if (DEVI_IS_ATTACHING(dip)) {
424                         USB_DPRINTF_L2(DPRINT_MASK_ATTA, hubdi_log_handle,
425                             "failure to unbind root hub after attach failure");
426                 }

428                 return (USB_FAILURE);
429         }

431         usba_device = usba_get_usba_device(dip);

433         kmem_free(usba_device->usb_root_hubd, sizeof (hubd_t));

435         kmem_free(usba_device->usb_cfg_array,
436             usba_device->usb_cfg_array_length);
437         kmem_free(usba_device->usb_cfg_array_len,
438             usba_device->usb_cfg_array_len_length);

440         kmem_free(usba_device->usb_cfg_str_descr, sizeof (uchar_t *));

442         usba_free_usba_device(usba_device);

444         usba_rem_root_hub(dip);

446         (void) ndi_prop_remove(DDI_DEV_T_NONE, dip, "root-hub");

448         return (USB_SUCCESS);
449 }


452 /*
453  * Actual Hub Driver support code:
454  *      shared by root hub and non-root hubs
455  */
456 #include <sys/usb/usba/usbai_version.h>
```



```
458 /* Debugging support */
459 uint_t hubd_errlevel    = USB_LOG_L4;
460 uint_t hubd_errmask     = (uint_t)DPRINT_MASK_ALL;
461 uint_t hubd_instance_debug = (uint_t)-1;
462 static uint_t hubdi_bus_config_debug = 0;

464 _NOTE(DATA_READABLE_WITHOUT_LOCK(hubd_errlevel))
465 _NOTE(DATA_READABLE_WITHOUT_LOCK(hubd_errmask))
466 _NOTE(DATA_READABLE_WITHOUT_LOCK(hubd_instance_debug))

468 _NOTE(SCHEME_PROTECTS_DATA("unique", msgb))
469 _NOTE(SCHEME_PROTECTS_DATA("unique", dev_info))


472 /*
473  * local variables:
474  *
475  * Amount of time to wait between resetting the port and accessing
476  * the device.  The value is in microseconds.
477  */
478 static uint_t hubd_device_delay = 1000000;

480 /*
481  * enumeration retry
482  */
483 #define HUBD_PORT_RETRY 5
484 static uint_t hubd_retry_enumerate = HUBD_PORT_RETRY;

486 /*
487  * Stale hotremoved device cleanup delay
488  */
489 #define HUBD_STALE_DIP_CLEANUP_DELAY    5000000
490 static uint_t hubd_dip_cleanup_delay = HUBD_STALE_DIP_CLEANUP_DELAY;

492 /*
493  * retries for USB suspend and resume
494  */
495 #define HUBD_SUS_RES_RETRY      2

497 void    *hubd_statep;

499 /*
500  * prototypes
501  */
502 static int hubd_cleanup(dev_info_t *dip, hubd_t  *hubd);
503 static int hubd_check_ports(hubd_t  *hubd);

505 static int  hubd_open_intr_pipe(hubd_t *hubd);
506 static void hubd_start_polling(hubd_t *hubd, int always);
507 static void hubd_stop_polling(hubd_t *hubd);
508 static void hubd_close_intr_pipe(hubd_t *hubd);

510 static void hubd_read_cb(usb_pipe_handle_t pipe, usb_intr_req_t *req);
511 static void hubd_exception_cb(usb_pipe_handle_t pipe,
512                                                 usb_intr_req_t *req);
513 static void hubd_hotplug_thread(void *arg);
514 static void hubd_reset_thread(void *arg);
515 static int hubd_create_child(dev_info_t *dip,
516                 hubd_t          *hubd,
517                 usba_device_t   *usba_device,
518                 usb_port_status_t port_status,
519                 usb_port_t      port,
520                 int             iteration);

522 static int hubd_delete_child(hubd_t *hubd, usb_port_t port, uint_t flag,
```

```
523         boolean_t retry);

525 static int hubd_get_hub_descriptor(hubd_t *hubd);

527 static int hubd_get_hub_status_words(hubd_t *hubd, uint16_t *status);

529 static int hubd_reset_port(hubd_t *hubd, usb_port_t port);

531 static int hubd_get_hub_status(hubd_t *hubd);

533 static int hubd_handle_port_connect(hubd_t *hubd, usb_port_t port);

535 static int hubd_disable_port(hubd_t *hubd, usb_port_t port);

537 static int hubd_enable_port(hubd_t *hubd, usb_port_t port);
538 static int hubd_recover_disabled_port(hubd_t *hubd, usb_port_t port);

540 static int hubd_determine_port_status(hubd_t *hubd, usb_port_t port,
541         uint16_t *status, uint16_t *change, uint_t ack_flag);

543 static int hubd_enable_all_port_power(hubd_t *hubd);
544 static int hubd_disable_all_port_power(hubd_t *hubd);
545 static int hubd_disable_port_power(hubd_t *hubd, usb_port_t port);
546 static int hubd_enable_port_power(hubd_t *hubd, usb_port_t port);

548 static void hubd_free_usba_device(hubd_t *hubd, usba_device_t *usba_device);

550 static int hubd_can_suspend(hubd_t *hubd);
551 static void hubd_restore_device_state(dev_info_t *dip, hubd_t *hubd);
552 static int hubd_setdevaddr(hubd_t *hubd, usb_port_t port);
553 static void hubd_setdevconfig(hubd_t *hubd, usb_port_t port);

555 static int hubd_register_events(hubd_t *hubd);
556 static void hubd_do_callback(hubd_t *hubd, dev_info_t *dip,
557         ddi_eventcookie_t cookie);
558 static void hubd_run_callbacks(hubd_t *hubd, usba_event_t type);
559 static void hubd_post_event(hubd_t *hubd, usb_port_t port, usba_event_t type);
560 static void hubd_create_pm_components(dev_info_t *dip, hubd_t *hubd);

562 static int hubd_disconnect_event_cb(dev_info_t *dip);
563 static int hubd_reconnect_event_cb(dev_info_t *dip);
564 static int hubd_pre_suspend_event_cb(dev_info_t *dip);
565 static int hubd_post_resume_event_cb(dev_info_t *dip);
566 static int hubd_cpr_suspend(hubd_t *hubd);
567 static void hubd_cpr_resume(dev_info_t *dip);
568 static int hubd_restore_state_cb(dev_info_t *dip);
569 static int hubd_check_same_device(hubd_t *hubd, usb_port_t port);

571 static int hubd_init_power_budget(hubd_t *hubd);

573 static ndi_event_definition_t hubd_ndi_event_defs[] = {
574         {USBA_EVENT_TAG_HOT_REMOVAL, DDI_DEVI_REMOVE_EVENT, EPL_KERNEL,
575                                             NDI_EVENT_POST_TO_ALL},
576         {USBA_EVENT_TAG_HOT_INSERTION, DDI_DEVI_INSERT_EVENT, EPL_KERNEL,
577                                             NDI_EVENT_POST_TO_ALL},
578         {USBA_EVENT_TAG_POST_RESUME, USBA_POST_RESUME_EVENT, EPL_KERNEL,
579                                             NDI_EVENT_POST_TO_ALL},
580         {USBA_EVENT_TAG_PRE_SUSPEND, USBA_PRE_SUSPEND_EVENT, EPL_KERNEL,
581                                             NDI_EVENT_POST_TO_ALL}
582 };

584 #define HUBD_N_NDI_EVENTS \
585         (sizeof (hubd_ndi_event_defs) / sizeof (ndi_event_definition_t))

587 static ndi_event_set_t hubd_ndi_events = {
588         NDI_EVENTS_REV1, HUBD_N_NDI_EVENTS, hubd_ndi_event_defs};
```



```
590 /* events received from parent */
591 static usb_event_t hubd_events = {
592         hubd_disconnect_event_cb,
593         hubd_reconnect_event_cb,
594         hubd_pre_suspend_event_cb,
595         hubd_post_resume_event_cb
596 };


599 /*
600  * hubd_get_soft_state() returns the hubd soft state
601  *
602  * WUSB support extends this function to support wire adapter class
603  * devices. The hubd soft state for the wire adapter class device
604  * would be stored in usb_root_hubd field of the usba_device structure,
605  * just as the USB host controller drivers do.
606  */
607 hubd_t *
608 hubd_get_soft_state(dev_info_t *dip)
609 {
610         if (dip == NULL) {

612                 return (NULL);
613         }

615         if (usba_is_root_hub(dip) || usba_is_wa(dip)) {
616                 usba_device_t *usba_device = usba_get_usba_device(dip);

618                 return (usba_device->usb_root_hubd);
619         } else {
620                 int instance = ddi_get_instance(dip);

622                 return (ddi_get_soft_state(hubd_statep, instance));
623         }
624 }


627 /*
628  * PM support functions:
629  */
630 /*ARGSUSED*/
631 static void
632 hubd_pm_busy_component(hubd_t *hubd, dev_info_t *dip, int component)
633 {
634         if (hubd->h_hubpm != NULL) {
635                 hubd->h_hubpm->hubp_busy_pm++;
636                 mutex_exit(HUBD_MUTEX(hubd));
637                 if (pm_busy_component(dip, 0) != DDI_SUCCESS) {
638                         mutex_enter(HUBD_MUTEX(hubd));
639                         hubd->h_hubpm->hubp_busy_pm--;
640                         mutex_exit(HUBD_MUTEX(hubd));
641                 }
642                 mutex_enter(HUBD_MUTEX(hubd));
643                 USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
644                     "hubd_pm_busy_component: %d", hubd->h_hubpm->hubp_busy_pm);
645         }
646 }


649 /*ARGSUSED*/
650 static void
651 hubd_pm_idle_component(hubd_t *hubd, dev_info_t *dip, int component)
652 {
653         if (hubd->h_hubpm != NULL) {
654                 mutex_exit(HUBD_MUTEX(hubd));
```

```
655                 if (pm_idle_component(dip, 0) == DDI_SUCCESS) {
656                         mutex_enter(HUBD_MUTEX(hubd));
657                         ASSERT(hubd->h_hubpm->hubp_busy_pm > 0);
658                         hubd->h_hubpm->hubp_busy_pm--;
659                         mutex_exit(HUBD_MUTEX(hubd));
660                 }
661                 mutex_enter(HUBD_MUTEX(hubd));
662                 USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
663                     "hubd_pm_idle_component: %d", hubd->h_hubpm->hubp_busy_pm);
664         }
665 }


668 /*
669  * track power level changes for children of this instance
670  */
671 static void
672 hubd_set_child_pwrlvl(hubd_t *hubd, usb_port_t port, uint8_t power)
673 {
674         int     old_power, new_power, pwr;
675         usb_port_t      portno;
676         hub_power_t     *hubpm;

678         USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
679             "hubd_set_child_pwrlvl: port=%d power=%d",
680             port, power);

682         mutex_enter(HUBD_MUTEX(hubd));
683         hubpm = hubd->h_hubpm;

685         old_power = 0;
686         for (portno = 1; portno <= hubd->h_hub_descr.bNbrPorts; portno++) {
687                 old_power += hubpm->hubp_child_pwrstate[portno];
688         }

690         /* assign the port power */
691         pwr = hubd->h_hubpm->hubp_child_pwrstate[port];
692         hubd->h_hubpm->hubp_child_pwrstate[port] = power;
693         new_power = old_power - pwr + power;

695         USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
696             "hubd_set_child_pwrlvl: new_power=%d old_power=%d",
697             new_power, old_power);

699         if ((new_power > 0) && (old_power == 0)) {
700                 /* we have the first child coming out of low power */
701                 (void) hubd_pm_busy_component(hubd, hubd->h_dip, 0);
702         } else if ((new_power == 0) && (old_power > 0)) {
703                 /* we have the last child going to low power */
704                 (void) hubd_pm_idle_component(hubd, hubd->h_dip, 0);
705         }
706         mutex_exit(HUBD_MUTEX(hubd));
707 }


710 /*
711  * given a child dip, locate its port number
712  */
713 static usb_port_t
714 hubd_child_dip2port(hubd_t *hubd, dev_info_t *dip)
715 {
716         usb_port_t      port;

718         mutex_enter(HUBD_MUTEX(hubd));
719         for (port = 1; port <= hubd->h_hub_descr.bNbrPorts; port++) {
720                 if (hubd->h_children_dips[port] == dip) {
```



```
722                         break;
723                 }
724         }
725         ASSERT(port <= hubd->h_hub_descr.bNbrPorts);
726         mutex_exit(HUBD_MUTEX(hubd));

728         return (port);
729 }


732 /*
733  * if the hub can be put into low power mode, return success
734  * NOTE: suspend here means going to lower power, not CPR suspend.
735  */
736 static int
737 hubd_can_suspend(hubd_t *hubd)
738 {
739         hub_power_t     *hubpm;
740         int             total_power = 0;
741         usb_port_t      port;

743         hubpm = hubd->h_hubpm;

745         if (DEVI_IS_DETACHING(hubd->h_dip)) {

747                 return (USB_SUCCESS);
748         }

750         /*
751          * Don't go to lower power if haven't been at full power for enough
752          * time to let hotplug thread kickoff.
753          */
754         if (gethrtime() < (hubpm->hubp_time_at_full_power +
 24         if (ddi_get_time() < (hubpm->hubp_time_at_full_power +
755             hubpm->hubp_min_pm_threshold)) {

757                 return (USB_FAILURE);
758         }

760         for (port = 1; (total_power == 0) &&
761             (port <= hubd->h_hub_descr.bNbrPorts); port++) {
762                 total_power += hubpm->hubp_child_pwrstate[port];
763         }

765         USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
766             "hubd_can_suspend: %d", total_power);

768         return (total_power ? USB_FAILURE : USB_SUCCESS);
769 }
______unchanged_portion_omitted_


1702 static int
1703 hubd_pwrlvl3(hubd_t *hubd)
1704 {
1705         hub_power_t     *hubpm;
1706         int             rval;

1708         USB_DPRINTF_L2(DPRINT_MASK_PM, hubd->h_log_handle, "hubd_pwrlvl3");

1710         hubpm = hubd->h_hubpm;
1711         switch (hubd->h_dev_state) {
1712         case USB_DEV_PWRED_DOWN:
1713                 ASSERT(hubpm->hubp_current_power == USB_DEV_OS_PWR_OFF);
1714                 if (usba_is_root_hub(hubd->h_dip)) {
```

```
1715                        /* implement global resume here */
1716                        USB_DPRINTF_L2(DPRINT_MASK_PM,
1717                            hubd->h_log_handle,
1718                            "Global Resume: Not Yet Implemented");
1719                }
1720                /* Issue USB D0 command to the device here */
1721                rval = usb_set_device_pwrlvl0(hubd->h_dip);
1722                ASSERT(rval == USB_SUCCESS);
1723                hubd->h_dev_state = USB_DEV_ONLINE;
1724                hubpm->hubp_current_power = USB_DEV_OS_FULL_PWR;
1725                hubpm->hubp_time_at_full_power = gethrtime();
 995                hubpm->hubp_time_at_full_power = ddi_get_time();
1726                hubd_start_polling(hubd, 0);

1728                /* FALLTHRU */
1729        case USB_DEV_ONLINE:
1730                /* we are already in full power */

1732                /* FALLTHRU */
1733        case USB_DEV_DISCONNECTED:
1734        case USB_DEV_SUSPENDED:
1735                /*
1736                 * PM framework tries to put you in full power
1737                 * during system shutdown. If we are disconnected
1738                 * return success. Also, we should not change state
1739                 * when we are disconnected or suspended or about to
1740                 * transition to that state
1741                 */

1743                return (USB_SUCCESS);
1744        default:
1745                USB_DPRINTF_L2(DPRINT_MASK_PM, hubd->h_log_handle,
1746                    "hubd_pwrlvl3: Illegal dev_state=%d", hubd->h_dev_state);

1748                return (USB_FAILURE);
1749        }
1750 }
```

*unchanged_portion_omitted_*

```
3643 /*
3644  * hubd_hotplug_thread:
3645  *      handles resetting of port, and creating children
3646  *
3647  *      the ports to check are indicated in h_port_change bit mask
3648  * XXX note that one time poll doesn't work on the root hub
3649  */
3650 static void
3651 hubd_hotplug_thread(void *arg)
3652 {
3653        hubd_hotplug_arg_t *hd_arg = (hubd_hotplug_arg_t *)arg;
3654        hubd_t          *hubd = hd_arg->hubd;
3655        boolean_t       attach_flg = hd_arg->hotplug_during_attach;
3656        usb_port_t      port;
3657        uint16_t        nports;
3658        uint16_t        status, change;
3659        hub_power_t     *hubpm;
3660        dev_info_t      *hdip = hubd->h_dip;
3661        dev_info_t      *rh_dip = hubd->h_usba_device->usb_root_hub_dip;
3662        dev_info_t      *child_dip;
3663        boolean_t       online_child = B_FALSE;
3664        boolean_t       offline_child = B_FALSE;
3665        boolean_t       pwrup_child = B_FALSE;
3666        int             prh_circ, rh_circ, chld_circ, circ, old_state;

3668        USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
```



```
3669            "hubd_hotplug_thread:  started");

3671        /*
3672         * Before console is init'd, we temporarily block the hotplug
3673         * threads so that BUS_CONFIG_ONE through hubd_bus_config() can be
3674         * processed quickly. This reduces the time needed for vfs_mountroot()
3675         * to mount the root FS from a USB disk. And on SPARC platform,
3676         * in order to load 'consconfig' successfully after OBP is gone,
3677         * we need to check 'modrootloaded' to make sure root filesystem is
3678         * available.
3679         */
3680        while (!modrootloaded || !consconfig_console_is_ready()) {
3681                delay(drv_usectohz(10000));
3682        }

3684        kmem_free(arg, sizeof (hubd_hotplug_arg_t));

3686        /*
3687         * if our bus power entry point is active, process the change
3688         * on the next notification of interrupt pipe
3689         */
3690        mutex_enter(HUBD_MUTEX(hubd));
3691        if (hubd->h_bus_pwr || (hubd->h_hotplug_thread > 1)) {
3692                hubd->h_hotplug_thread--;

3694                /* mark this device as idle */
3695                hubd_pm_idle_component(hubd, hubd->h_dip, 0);
3696                mutex_exit(HUBD_MUTEX(hubd));

3698                USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3699                    "hubd_hotplug_thread: "
3700                    "bus_power in progress/hotplugging undesirable - quit");

3702                return;
3703        }
3704        mutex_exit(HUBD_MUTEX(hubd));

3706        ndi_hold_devi(hdip); /* so we don't race with detach */

3708        mutex_enter(HUBD_MUTEX(hubd));

3710        /* is this the root hub? */
3711        if (hdip == rh_dip) {
3712                if (hubd->h_dev_state == USB_DEV_PWRED_DOWN) {
3713                        hubpm = hubd->h_hubpm;

3715                        /* mark the root hub as full power */
3716                        hubpm->hubp_current_power = USB_DEV_OS_FULL_PWR;
3717                        hubpm->hubp_time_at_full_power = gethrtime();
2987                        hubpm->hubp_time_at_full_power = ddi_get_time();
3718                        mutex_exit(HUBD_MUTEX(hubd));

3720                        USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3721                            "hubd_hotplug_thread: call pm_power_has_changed");

3723                        (void) pm_power_has_changed(hdip, 0,
3724                            USB_DEV_OS_FULL_PWR);

3726                        mutex_enter(HUBD_MUTEX(hubd));
3727                        hubd->h_dev_state = USB_DEV_ONLINE;
3728                }

3730        } else {
3731                USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3732                    "hubd_hotplug_thread: not root hub");
3733        }
```

```
3735         mutex_exit(HUBD_MUTEX(hubd));

3737         /*
3738          * this ensures one hotplug activity per system at a time.
3739          * we enter the parent PCI node to have this serialization.
3740          * this also excludes ioctls and deathrow thread
3741          * (a bit crude but easier to debug)
3742          */
3743         ndi_devi_enter(ddi_get_parent(rh_dip), &prh_circ);
3744         ndi_devi_enter(rh_dip, &rh_circ);

3746         /* exclude other threads */
3747         ndi_devi_enter(hdip, &circ);
3748         mutex_enter(HUBD_MUTEX(hubd));

3750         ASSERT(hubd->h_intr_pipe_state == HUBD_INTR_PIPE_ACTIVE);

3752         nports = hubd->h_hub_descr.bNbrPorts;

3754         hubd_stop_polling(hubd);

3756         while ((hubd->h_dev_state == USB_DEV_ONLINE) &&
3757             (hubd->h_port_change)) {
3758                 /*
3759                  * The 0th bit is the hub status change bit.
3760                  * handle loss of local power here
3761                  */
3762                 if (hubd->h_port_change & HUB_CHANGE_STATUS) {
3763                         USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3764                             "hubd_hotplug_thread: hub status change!");

3766                         /*
3767                          * This should be handled properly.  For now,
3768                          * mask off the bit.
3769                          */
3770                         hubd->h_port_change &= ~HUB_CHANGE_STATUS;

3772                         /*
3773                          * check and ack hub status
3774                          * this causes stall conditions
3775                          * when local power is removed
3776                          */
3777                         (void) hubd_get_hub_status(hubd);
3778                 }

3780                 for (port = 1; port <= nports; port++) {
3781                         usb_port_mask_t port_mask;
3782                         boolean_t was_connected;

3784                         port_mask = 1 << port;
3785                         was_connected =
3786                             (hubd->h_port_state[port] & PORT_STATUS_CCS) &&
3787                             (hubd->h_children_dips[port]);

3789                         USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3790                             "hubd_hotplug_thread: "
3791                             "port %d mask=0x%x change=0x%x connected=0x%x",
3792                             port, port_mask, hubd->h_port_change,
3793                             was_connected);

3795                         /*
3796                          * is this a port connection that changed?
3797                          */
3798                         if ((hubd->h_port_change & port_mask) == 0) {
```



```
3800                                 continue;
3801                         }
3802                         hubd->h_port_change &= ~port_mask;

3804                         /* ack all changes */
3805                         (void) hubd_determine_port_status(hubd, port,
3806                             &status, &change, HUBD_ACK_ALL_CHANGES);

3808                         USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3809                             "handle port %d:\n\t"
3810                             "new status=0x%x change=0x%x was_conn=0x%x ",
3811                             port, status, change, was_connected);

3813                         /* Recover a disabled port */
3814                         if (change & PORT_CHANGE_PESC) {
3815                                 USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG,
3816                                     hubd->h_log_handle,
3817                                     "port%d Disabled - "
3818                                     "status=0x%x, change=0x%x",
3819                                     port, status, change);

3821                                 /*
3822                                  * if the port was connected and is still
3823                                  * connected, recover the port
3824                                  */
3825                                 if (was_connected && (status &
3826                                     PORT_STATUS_CCS)) {
3827                                         online_child |=
3828                                             (hubd_recover_disabled_port(hubd,
3829                                             port) == USB_SUCCESS);
3830                                 }
3831                         }

3833                         /*
3834                          * Now check what changed on the port
3835                          */
3836                         if ((change & PORT_CHANGE_CSC) || attach_flg) {
3837                                 if ((status & PORT_STATUS_CCS) &&
3838                                     (!was_connected)) {
3839                                         /* new device plugged in */
3840                                         online_child |=
3841                                             (hubd_handle_port_connect(hubd,
3842                                             port) == USB_SUCCESS);

3844                                 } else if ((status & PORT_STATUS_CCS) &&
3845                                     was_connected) {
3846                                         /*
3847                                          * In this case we can never be sure
3848                                          * if the device indeed got hotplugged
3849                                          * or the hub is falsely reporting the
3850                                          * change.
3851                                          */
3852                                         child_dip = hubd->h_children_dips[port];

3854                                         mutex_exit(HUBD_MUTEX(hubd));
3855                                         /*
3856                                          * this ensures we do not race with
3857                                          * other threads which are detaching
3858                                          * the child driver at the same time.
3859                                          */
3860                                         ndi_devi_enter(child_dip, &chld_circ);
3861                                         /*
3862                                          * Now check if the driver remains
3863                                          * attached.
3864                                          */
3865                                         if (i_ddi_devi_attached(child_dip)) {
```

```
3866                                            /*
3867                                             * first post a disconnect event
3868                                             * to the child.
3869                                             */
3870                                            hubd_post_event(hubd, port,
3871                                                USBA_EVENT_TAG_HOT_REMOVAL);
3872                                            mutex_enter(HUBD_MUTEX(hubd));

3874                                            /*
3875                                             * then reset the port and
3876                                             * recover the device
3877                                             */
3878                                            online_child |=
3879                                                (hubd_handle_port_connect(
3880                                                hubd, port) == USB_SUCCESS);

3882                                            mutex_exit(HUBD_MUTEX(hubd));
3883                                    }

3885                                    ndi_devi_exit(child_dip, chld_circ);
3886                                    mutex_enter(HUBD_MUTEX(hubd));
3887                            } else if (was_connected) {
3888                                    /* this is a disconnect */
3889                                    mutex_exit(HUBD_MUTEX(hubd));
3890                                    hubd_post_event(hubd, port,
3891                                        USBA_EVENT_TAG_HOT_REMOVAL);
3892                                    mutex_enter(HUBD_MUTEX(hubd));

3894                                    offline_child = B_TRUE;
3895                            }
3896                    }

3898                    /*
3899                     * Check if any port is coming out of suspend
3900                     */
3901                    if (change & PORT_CHANGE_PSSC) {
3902                            /* a resuming device could have disconnected */
3903                            if (was_connected &&
3904                                hubd->h_children_dips[port]) {

3906                                    /* device on this port resuming */
3907                                    dev_info_t *dip;

3909                                    dip = hubd->h_children_dips[port];

3911                                    /*
3912                                     * Don't raise power on detaching child
3913                                     */
3914                                    if (!DEVI_IS_DETACHING(dip)) {
3915                                            /*
3916                                             * As this child is not
3917                                             * detaching, we set this
3918                                             * flag, causing bus_ctls
3919                                             * to stall detach till
3920                                             * pm_raise_power returns
3921                                             * and flag it for a deferred
3922                                             * raise_power.
3923                                             *
3924                                             * pm_raise_power is deferred
3925                                             * because we need to release
3926                                             * the locks first.
3927                                             */
3928                                            hubd->h_port_state[port] |=
3929                                                HUBD_CHILD_RAISE_POWER;
3930                                            pwrup_child = B_TRUE;
3931                                            mutex_exit(HUBD_MUTEX(hubd));
```



```
3933                                            /*
3934                                             * make sure that child
3935                                             * doesn't disappear
3936                                             */
3937                                            ndi_hold_devi(dip);

3939                                            mutex_enter(HUBD_MUTEX(hubd));
3940                                    }
3941                            }
3942                    }

3944                    /*
3945                     * Check if the port is over-current
3946                     */
3947                    if (change & PORT_CHANGE_OCIC) {
3948                            USB_DPRINTF_L1(DPRINT_MASK_HOTPLUG,
3949                                hubd->h_log_handle,
3950                                "Port%d in over current condition, "
3951                                "please check the attached device to "
3952                                "clear the condition. The system will "
3953                                "try to recover the port, but if not "
3954                                "successful, you need to re-connect "
3955                                "the hub or reboot the system to bring "
3956                                "the port back to work", port);

3958                            if (!(status & PORT_STATUS_PPS)) {
3959                                    /*
3960                                     * Try to enable port power, but
3961                                     * possibly fail. Ignore failure
3962                                     */
3963                                    (void) hubd_enable_port_power(hubd,
3964                                        port);

3966                                    /*
3967                                     * Delay some time to avoid
3968                                     * over-current event to happen
3969                                     * too frequently in some cases
3970                                     */
3971                                    mutex_exit(HUBD_MUTEX(hubd));
3972                                    delay(drv_usectohz(500000));
3973                                    mutex_enter(HUBD_MUTEX(hubd));
3974                            }
3975                    }
3976            }
3977    }

3979    /* release locks so we can do a devfs_clean */
3980    mutex_exit(HUBD_MUTEX(hubd));

3982    /* delete cached dv_node's but drop locks first */
3983    ndi_devi_exit(hdip, circ);
3984    ndi_devi_exit(rh_dip, rh_circ);
3985    ndi_devi_exit(ddi_get_parent(rh_dip), prh_circ);

3987    (void) devfs_clean(rh_dip, NULL, 0);

3989    /* now check if any children need onlining */
3990    if (online_child) {
3991            USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
3992                "hubd_hotplug_thread: onlining children");

3994            (void) ndi_devi_online(hubd->h_dip, 0);
3995    }

3997    /* now check if any disconnected devices need to be cleaned up */
```

```
3998         if (offline_child) {
3999                 USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
4000                     "hubd_hotplug_thread: scheduling cleanup");
4002                 hubd_schedule_cleanup(hubd->h_usba_device->usb_root_hub_dip);
4003         }
4005         mutex_enter(HUBD_MUTEX(hubd));
4007         /* now raise power on the children that have woken up */
4008         if (pwrup_child) {
4009                 old_state = hubd->h_dev_state;
4010                 hubd->h_dev_state = USB_DEV_HUB_CHILD_PWRLVL;
4011                 for (port = 1; port <= nports; port++) {
4012                         if (hubd->h_port_state[port] & HUBD_CHILD_RAISE_POWER) {
4013                                 dev_info_t *dip = hubd->h_children_dips[port];
4015                                 mutex_exit(HUBD_MUTEX(hubd));
4017                                 /* Get the device to full power */
4018                                 (void) pm_busy_component(dip, 0);
4019                                 (void) pm_raise_power(dip, 0,
4020                                     USB_DEV_OS_FULL_PWR);
4021                                 (void) pm_idle_component(dip, 0);
4023                                 /* release the hold on the child */
4024                                 ndi_rele_devi(dip);
4025                                 mutex_enter(HUBD_MUTEX(hubd));
4026                                 hubd->h_port_state[port] &=
4027                                     ~HUBD_CHILD_RAISE_POWER;
4028                         }
4029                 }
4030                 /*
4031                  * make sure that we don't accidentally
4032                  * over write the disconnect state
4033                  */
4034                 if (hubd->h_dev_state == USB_DEV_HUB_CHILD_PWRLVL) {
4035                         hubd->h_dev_state = old_state;
4036                 }
4037         }
4039         /*
4040          * start polling can immediately kick off read callback
4041          * we need to set the h_hotplug_thread to 0 so that
4042          * the callback is not dropped
4043          *
4044          * if there is device during reset, still stop polling to avoid the
4045          * read callback interrupting the reset, the polling will be started
4046          * in hubd_reset_thread.
4047          */
4048         for (port = 1; port <= MAX_PORTS; port++) {
4049                 if (hubd->h_reset_port[port]) {
4051                         break;
4052                 }
4053         }
4054         if (port > MAX_PORTS) {
4055                 hubd_start_polling(hubd, HUBD_ALWAYS_START_POLLING);
4056         }
4058         /*
4059          * Earlier we would set the h_hotplug_thread = 0 before
4060          * polling was restarted  so that
4061          * if there is any root hub status change interrupt, we can still kick
4062          * off the hotplug thread. This was valid when this interrupt was
4063          * delivered in hardware, and only ONE interrupt would be delivered.
```



```
4064          * Now that we poll on the root hub looking for status change in
4065          * software, this assignment is no longer required.
4066          */
4067         hubd->h_hotplug_thread--;
4069         /* mark this device as idle */
4070         (void) hubd_pm_idle_component(hubd, hubd->h_dip, 0);
4072         cv_broadcast(&hubd->h_cv_hotplug_dev);
4074         USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
4075             "hubd_hotplug_thread: exit");
4077         mutex_exit(HUBD_MUTEX(hubd));
4079         ndi_rele_devi(hdip);
4080 }
_____unchanged_portion_omitted_


7258 /*
7259  * Power management
7260  *
7261  * create the pm components required for power management
7262  */
7263 static void
7264 hubd_create_pm_components(dev_info_t *dip, hubd_t *hubd)
7265 {
7266         hub_power_t     *hubpm;
7268         USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
7269             "hubd_create_pm_components: Begin");
7271         /* Allocate the state structure */
7272         hubpm = kmem_zalloc(sizeof (hub_power_t), KM_SLEEP);
7274         hubd->h_hubpm = hubpm;
7275         hubpm->hubp_hubd = hubd;
7276         hubpm->hubp_pm_capabilities = 0;
7277         hubpm->hubp_current_power = USB_DEV_OS_FULL_PWR;
7278         hubpm->hubp_time_at_full_power = gethrtime();
7279         hubpm->hubp_min_pm_threshold = hubdi_min_pm_threshold * NANOSEC;
6548         hubpm->hubp_time_at_full_power = ddi_get_time();
6549         hubpm->hubp_min_pm_threshold = hubdi_min_pm_threshold;
7281         /* alloc memory to save power states of children */
7282         hubpm->hubp_child_pwrstate = (uint8_t *)
7283             kmem_zalloc(MAX_PORTS + 1, KM_SLEEP);
7285         /*
7286          * if the enable remote wakeup fails
7287          * we still want to enable
7288          * parent notification so we can PM the children
7289          */
7290         usb_enable_parent_notification(dip);
7292         if (usb_handle_remote_wakeup(dip,
7293             USB_REMOTE_WAKEUP_ENABLE) == USB_SUCCESS) {
7294                 uint_t          pwr_states;
7296                 USB_DPRINTF_L2(DPRINT_MASK_PM, hubd->h_log_handle,
7297                     "hubd_create_pm_components: "
7298                     "Remote Wakeup Enabled");
7300                 if (usb_create_pm_components(dip, &pwr_states) ==
7301                     USB_SUCCESS) {
```

```
7302                        mutex_enter(HUBD_MUTEX(hubd));
7303                        hubpm->hubp_wakeup_enabled = 1;
7304                        hubpm->hubp_pwr_states = (uint8_t)pwr_states;

7306                        /* we are busy now till end of the attach */
7307                        hubd_pm_busy_component(hubd, dip, 0);
7308                        mutex_exit(HUBD_MUTEX(hubd));

7310                        /* bring the device to full power */
7311                        (void) pm_raise_power(dip, 0,
7312                            USB_DEV_OS_FULL_PWR);
7313                }
7314        }

7316        USB_DPRINTF_L4(DPRINT_MASK_PM, hubd->h_log_handle,
7317            "hubd_create_pm_components: END");
7318 }
```

*unchanged_portion_omitted_*

```
8647 /*
8648  * hubd_reset_thread:
8649  *      handles the "USB_RESET_LVL_REATTACH" reset of usb device.
8650  *
8651  *      - delete the child (force detaching the device and its children)
8652  *      - reset the corresponding parent hub port
8653  *      - create the child (force re-attaching the device and its children)
8654  */
8655 static void
8656 hubd_reset_thread(void *arg)
8657 {
8658        hubd_reset_arg_t *hd_arg = (hubd_reset_arg_t *)arg;
8659        hubd_t           *hubd = hd_arg->hubd;
8660        uint16_t         reset_port = hd_arg->reset_port;
8661        uint16_t         status, change;
8662        hub_power_t      *hubpm;
8663        dev_info_t       *hdip = hubd->h_dip;
8664        dev_info_t       *rh_dip = hubd->h_usba_device->usb_root_hub_dip;
8665        dev_info_t       *child_dip;
8666        boolean_t        online_child = B_FALSE;
8667        int              prh_circ, rh_circ, circ, devinst;
8668        char             *devname;
8669        int              i = 0;
8670        int              rval = USB_FAILURE;

8672        USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8673            "hubd_reset_thread:  started, hubd_reset_port = 0x%x", reset_port);

8675        kmem_free(arg, sizeof (hubd_reset_arg_t));

8677        mutex_enter(HUBD_MUTEX(hubd));

8679        child_dip = hubd->h_children_dips[reset_port];
8680        ASSERT(child_dip != NULL);

8682        devname = (char *)ddi_driver_name(child_dip);
8683        devinst = ddi_get_instance(child_dip);

8685        /* if our bus power entry point is active, quit the reset */
8686        if (hubd->h_bus_pwr) {
8687                USB_DPRINTF_L0(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8688                    "%s%d is under bus power management, cannot be reset. "
8689                    "Please disconnect and reconnect this device.",
8690                    devname, devinst);

8692                goto Fail;
8693        }
```



```
8695        if (hubd_wait_for_hotplug_exit(hubd) == USB_FAILURE) {
8696                /* we got woken up because of a timeout */
8697                USB_DPRINTF_L0(DPRINT_MASK_HOTPLUG,
8698                    hubd->h_log_handle, "Time out when resetting the device"
8699                    " %s%d. Please disconnect and reconnect this device.",
8700                    devname, devinst);

8702                goto Fail;
8703        }

8705        hubd->h_hotplug_thread++;

8707        /* is this the root hub? */
8708        if ((hdip == rh_dip) &&
8709            (hubd->h_dev_state == USB_DEV_PWRED_DOWN)) {
8710                hubpm = hubd->h_hubpm;

8712                /* mark the root hub as full power */
8713                hubpm->hubp_current_power = USB_DEV_OS_FULL_PWR;
8714                hubpm->hubp_time_at_full_power = gethrtime();
7984                hubpm->hubp_time_at_full_power = ddi_get_time();
8715                mutex_exit(HUBD_MUTEX(hubd));

8717                USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8718                    "hubd_reset_thread: call pm_power_has_changed");

8720                (void) pm_power_has_changed(hdip, 0,
8721                    USB_DEV_OS_FULL_PWR);

8723                mutex_enter(HUBD_MUTEX(hubd));
8724                hubd->h_dev_state = USB_DEV_ONLINE;
8725        }

8727        mutex_exit(HUBD_MUTEX(hubd));

8729        /*
8730         * this ensures one reset activity per system at a time.
8731         * we enter the parent PCI node to have this serialization.
8732         * this also excludes ioctls and deathrow thread
8733         */
8734        ndi_devi_enter(ddi_get_parent(rh_dip), &prh_circ);
8735        ndi_devi_enter(rh_dip, &rh_circ);

8737        /* exclude other threads */
8738        ndi_devi_enter(hdip, &circ);
8739        mutex_enter(HUBD_MUTEX(hubd));

8741        /*
8742         * We need to make sure that the child is still online for a hotplug
8743         * thread could have inserted which detached the child.
8744         */
8745        if (hubd->h_children_dips[reset_port]) {
8746                mutex_exit(HUBD_MUTEX(hubd));
8747                /* First disconnect the device */
8748                hubd_post_event(hubd, reset_port, USBA_EVENT_TAG_HOT_REMOVAL);

8750                /* delete cached dv_node's but drop locks first */
8751                ndi_devi_exit(hdip, circ);
8752                ndi_devi_exit(rh_dip, rh_circ);
8753                ndi_devi_exit(ddi_get_parent(rh_dip), prh_circ);

8755                (void) devfs_clean(rh_dip, NULL, DV_CLEAN_FORCE);

8757                /*
8758                 * workaround only for storage device. When it's able to force
```

```
8759                 * detach a driver, this code can be removed safely.
8760                 *
8761                 * If we're to reset storage device and the device is used, we
8762                 * will wait at most extra 20s for applications to exit and
8763                 * close the device. This is especially useful for HAL-based
8764                 * applications.
8765                 */
8766                if ((strcmp(devname, "scsa2usb") == 0) &&
8767                    DEVI(child_dip)->devi_ref != 0) {
8768                        while (i++ < hubdi_reset_delay) {
8769                                mutex_enter(HUBD_MUTEX(hubd));
8770                                rval = hubd_delete_child(hubd, reset_port,
8771                                    NDI_DEVI_REMOVE, B_FALSE);
8772                                mutex_exit(HUBD_MUTEX(hubd));
8773                                if (rval == USB_SUCCESS)
8774                                        break;

8776                                delay(drv_usectohz(1000000)); /* 1s */
8777                        }
8778                }

8780                ndi_devi_enter(ddi_get_parent(rh_dip), &prh_circ);
8781                ndi_devi_enter(rh_dip, &rh_circ);
8782                ndi_devi_enter(hdip, &circ);

8784                mutex_enter(HUBD_MUTEX(hubd));

8786                /* Then force detaching the device */
8787                if ((rval != USB_SUCCESS) && (hubd_delete_child(hubd,
8788                    reset_port, NDI_DEVI_REMOVE, B_FALSE) != USB_SUCCESS)) {
8789                        USB_DPRINTF_L0(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8790                            "%s%d cannot be reset due to other applications "
8791                            "are using it, please first close these "
8792                            "applications, then disconnect and reconnect"
8793                            "the device.", devname, devinst);

8795                        mutex_exit(HUBD_MUTEX(hubd));
8796                        /* post a re-connect event */
8797                        hubd_post_event(hubd, reset_port,
8798                            USBA_EVENT_TAG_HOT_INSERTION);
8799                        mutex_enter(HUBD_MUTEX(hubd));
8800                } else {
8801                        (void) hubd_determine_port_status(hubd, reset_port,
8802                            &status, &change, HUBD_ACK_ALL_CHANGES);

8804                        /* Reset the parent hubd port and create new child */
8805                        if (status & PORT_STATUS_CCS) {
8806                                online_child |= (hubd_handle_port_connect(hubd,
8807                                    reset_port) == USB_SUCCESS);
8808                        }
8809                }
8810        }

8812        /* release locks so we can do a devfs_clean */
8813        mutex_exit(HUBD_MUTEX(hubd));

8815        /* delete cached dv_node's but drop locks first */
8816        ndi_devi_exit(hdip, circ);
8817        ndi_devi_exit(rh_dip, rh_circ);
8818        ndi_devi_exit(ddi_get_parent(rh_dip), prh_circ);

8820        (void) devfs_clean(rh_dip, NULL, 0);

8822        /* now check if any children need onlining */
8823        if (online_child) {
8824                USB_DPRINTF_L3(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
```



```
8825                    "hubd_reset_thread: onlining children");

8827                (void) ndi_devi_online(hubd->h_dip, 0);
8828        }

8830        mutex_enter(HUBD_MUTEX(hubd));

8832        /* allow hotplug thread now */
8833        hubd->h_hotplug_thread--;
8834 Fail:
8835        hubd_start_polling(hubd, 0);

8837        /* mark this device as idle */
8838        (void) hubd_pm_idle_component(hubd, hubd->h_dip, 0);

8840        USB_DPRINTF_L4(DPRINT_MASK_HOTPLUG, hubd->h_log_handle,
8841            "hubd_reset_thread: exit, %d", hubd->h_hotplug_thread);

8843        hubd->h_reset_port[reset_port] = B_FALSE;

8845        mutex_exit(HUBD_MUTEX(hubd));

8847        ndi_rele_devi(hdip);
8848 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   12448 Mon May  5 11:11:19 2014
new/usr/src/uts/common/sys/usb/hubd/hubdvar.h
4782 usba shouldn't abuse ddi_get_time(9f)
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */
#endif /* ! codereview */

#ifndef _SYS_USB_HUBDVAR_H
#define _SYS_USB_HUBDVAR_H


#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/sunndi.h>
#include <sys/ndi_impldefs.h>
#include <sys/usb/usba/usba_types.h>
#include <sys/callb.h>

/*
 * HUB USB device state management :
 *
 *                          CHILD_PWRLVL---1>--------+
 *                              ^                    |
 *                              8                    |
 *                              |                    |
 *                              9                    |
 *                              v                    |
 *      PWRED_DWN---<3----4>--ONLINE---<2-----1>-DISCONNECTED
 *          |                 |  ^                 |  |
 *          |                 |  10                |  |
 *          |                 |  |                 |  |
 *          |                 |  RECOVER-<2-------+  |
 *          |                 |  ^                   |
 *          |                5  6                   |
 *          |                v  |                   |
```



```
 *          +----5>----------SUSPENDED----<5----7>----+
 *
 *      1 = Device Unplug
 *      2 = Original Device reconnected and after hub driver restores its own
 *          device state.
 *      3 = Device idles for time T & transitions to low power state
 *      4 = Remote wakeup by device OR Application kicking off IO to device
 *      5 = Notification to save state prior to DDI_SUSPEND
 *      6 = Notification to restore state after DDI_RESUME with correct device
 *          and after hub driver restores its own device state.
 *      7 = Notification to restore state after DDI_RESUME with device
 *          disconnected or a wrong device
 *      8 = Hub detect child doing remote wakeup and request the PM
 *          framework to bring it to full power
 *      9 = PM framework has compeleted call power entry point of the child
 *          and bus ctls of hub
 *     10 = Restoring states of its children i.e. set addrs & config.
 *
 */

#define HUBD_INITIAL_SOFT_SPACE 4

typedef struct hub_power_struct {
        void            *hubp_hubd;     /* points back to hubd_t */

        uint8_t         hubp_wakeup_enabled;    /* remote wakeup enabled? */

        /* this is the bit mask of the power states that device has */
        uint8_t         hubp_pwr_states;

        int             hubp_busy_pm;   /* device busy accounting */

        /* wakeup and power transition capabilities of an interface */
        uint8_t         hubp_pm_capabilities;

        uint8_t         hubp_current_power;     /* current power level */

        hrtime_t        hubp_time_at_full_power;        /* timestamp 0->3 */
        time_t          hubp_time_at_full_power;        /* timestamp 0->3 */

        hrtime_t        hubp_min_pm_threshold;          /* in nanoseconds */
        uint8_t         hubp_min_pm_threshold;          /* in seconds */

        /* power state of all children are tracked here */
        uint8_t         *hubp_child_pwrstate;

        /* pm-components properties are stored here */
        char            *hubp_pmcomp[5];

        usba_cfg_pwr_descr_t    hubp_confpwr_descr; /* config pwr descr */
} hub_power_t;
______unchanged_portion_omitted_
```