```
**********************************************************
   28786 Mon Jan 28 16:02:58 2013
new/usr/src/uts/common/io/usb/usba/hcdi.c
3503 usba doesn't initialize cc_no_resources, causes gibberish output in kstat:
**********************************************************
______unchanged_portion_omitted_


 266 /*
 267  * Allocate the hotplug kstats structure
 268  */
 269 void
 270 usba_hcdi_create_stats(usba_hcdi_t *hcdi, int instance)
 271 {
 272         char                    kstatname[KSTAT_STRLEN];
 273         const char              *dname = ddi_driver_name(hcdi->hcdi_dip);
 274         hcdi_hotplug_stats_t    *hsp;
 275         hcdi_error_stats_t      *esp;

 277         if (HCDI_HOTPLUG_STATS(hcdi) == NULL) {
 278                 (void) snprintf(kstatname, KSTAT_STRLEN, "%s%d,hotplug",
 279                     dname, instance);
 280                 HCDI_HOTPLUG_STATS(hcdi) = kstat_create("usba", instance,
 281                     kstatname, "usb_hotplug", KSTAT_TYPE_NAMED,
 282                     sizeof (hcdi_hotplug_stats_t) / sizeof (kstat_named_t),
 283                     KSTAT_FLAG_PERSISTENT);

 285                 if (HCDI_HOTPLUG_STATS(hcdi) == NULL) {

 287                         return;
 288                 }

 290                 hsp = HCDI_HOTPLUG_STATS_DATA(hcdi);
 291                 kstat_named_init(&hsp->hcdi_hotplug_total_success,
 292                     "Total Hotplug Successes", KSTAT_DATA_UINT64);
 293                 kstat_named_init(&hsp->hcdi_hotplug_success,
 294                     "Hotplug Successes", KSTAT_DATA_UINT64);
 295                 kstat_named_init(&hsp->hcdi_hotplug_total_failure,
 296                     "Hotplug Total Failures", KSTAT_DATA_UINT64);
 297                 kstat_named_init(&hsp->hcdi_hotplug_failure,
 298                     "Hotplug Failures", KSTAT_DATA_UINT64);
 299                 kstat_named_init(&hsp->hcdi_device_count,
 300                     "Device Count", KSTAT_DATA_UINT64);

 302                 HCDI_HOTPLUG_STATS(hcdi)->ks_private = hcdi;
 303                 HCDI_HOTPLUG_STATS(hcdi)->ks_update = nulldev;
 304                 kstat_install(HCDI_HOTPLUG_STATS(hcdi));
 305         }

 307         if (HCDI_ERROR_STATS(hcdi) == NULL) {
 308                 (void) snprintf(kstatname, KSTAT_STRLEN, "%s%d,error",
 309                     dname, instance);
 310                 HCDI_ERROR_STATS(hcdi) = kstat_create("usba", instance,
 311                     kstatname, "usb_errors", KSTAT_TYPE_NAMED,
 312                     sizeof (hcdi_error_stats_t) / sizeof (kstat_named_t),
 313                     KSTAT_FLAG_PERSISTENT);

 315                 if (HCDI_ERROR_STATS(hcdi) == NULL) {

 317                         return;
 318                 }

 320                 esp = HCDI_ERROR_STATS_DATA(hcdi);
 321                 kstat_named_init(&esp->cc_crc, "CRC Errors", KSTAT_DATA_UINT64);
 322                 kstat_named_init(&esp->cc_bitstuffing,
 323                     "Bit Stuffing Violations", KSTAT_DATA_UINT64);
```



```
 324                 kstat_named_init(&esp->cc_data_toggle_mm,
 325                     "Data Toggle PID Errors", KSTAT_DATA_UINT64);
 326                 kstat_named_init(&esp->cc_stall,
 327                     "Endpoint Stalls", KSTAT_DATA_UINT64);
 328                 kstat_named_init(&esp->cc_dev_not_resp,
 329                     "Device Not Responding", KSTAT_DATA_UINT64);
 330                 kstat_named_init(&esp->cc_pid_checkfailure,
 331                     "PID Check Bit Errors", KSTAT_DATA_UINT64);
 332                 kstat_named_init(&esp->cc_unexp_pid,
 333                     "Invalid PID Errors", KSTAT_DATA_UINT64);
 334                 kstat_named_init(&esp->cc_data_overrun,
 335                     "Data Overruns", KSTAT_DATA_UINT64);
 336                 kstat_named_init(&esp->cc_data_underrun,
 337                     "Data Underruns", KSTAT_DATA_UINT64);
 338                 kstat_named_init(&esp->cc_buffer_overrun,
 339                     "Buffer Overruns", KSTAT_DATA_UINT64);
 340                 kstat_named_init(&esp->cc_buffer_underrun,
 341                     "Buffer Underruns", KSTAT_DATA_UINT64);
 342                 kstat_named_init(&esp->cc_timeout,
 343                     "Command Timed Out", KSTAT_DATA_UINT64);
 344                 kstat_named_init(&esp->cc_not_accessed,
 345                     "Not Accessed By Hardware", KSTAT_DATA_UINT64);
 346                 kstat_named_init(&esp->cc_no_resources,
 347                     "No Resources", KSTAT_DATA_UINT64);
 348 #endif /* ! codereview */
 349                 kstat_named_init(&esp->cc_unspecified_err,
 350                     "Unspecified Error", KSTAT_DATA_UINT64);
 351                 kstat_named_init(&esp->cc_stopped_polling,
 352                     "Stopped Polling", KSTAT_DATA_UINT64);
 353                 kstat_named_init(&esp->cc_pipe_closing,
 354                     "Pipe Closing", KSTAT_DATA_UINT64);
 355                 kstat_named_init(&esp->cc_pipe_reset,
 356                     "Pipe Reset", KSTAT_DATA_UINT64);
 357                 kstat_named_init(&esp->cc_not_supported,
 358                     "Command Not Supported", KSTAT_DATA_UINT64);
 359                 kstat_named_init(&esp->cc_flushed,
 360                     "Request Flushed", KSTAT_DATA_UINT64);
 361 #endif /* ! codereview */
 362 #ifdef  NOTYETNEEDED
 363                 kstat_named_init(&esp->hcdi_usb_failure,
 364                     "USB Failure", KSTAT_DATA_UINT64);
 365                 kstat_named_init(&esp->hcdi_usb_no_resources,
 366                     "No Resources", KSTAT_DATA_UINT64);
 367                 kstat_named_init(&esp->hcdi_usb_no_bandwidth,
 368                     "No Bandwidth", KSTAT_DATA_UINT64);
 369                 kstat_named_init(&esp->hcdi_usb_pipe_reserved,
 370                     "Pipe Reserved", KSTAT_DATA_UINT64);
 371                 kstat_named_init(&esp->hcdi_usb_pipe_unshareable,
 372                     "Pipe Unshareable", KSTAT_DATA_UINT64);
 373                 kstat_named_init(&esp->hcdi_usb_not_supported,
 374                     "Function Not Supported", KSTAT_DATA_UINT64);
 375                 kstat_named_init(&esp->hcdi_usb_pipe_error,
 376                     "Pipe Error", KSTAT_DATA_UINT64);
 377                 kstat_named_init(&esp->hcdi_usb_pipe_busy,
 378                     "Pipe Busy", KSTAT_DATA_UINT64);
 379 #endif

 381                 HCDI_ERROR_STATS(hcdi)->ks_private = hcdi;
 382                 HCDI_ERROR_STATS(hcdi)->ks_update = nulldev;
 383                 kstat_install(HCDI_ERROR_STATS(hcdi));
 384         }
 385 }


 388 /*
 389  * Do actual error stats
```

```
 */
void
usba_hcdi_update_error_stats(usba_hcdi_t *hcdi, usb_cr_t completion_reason)
{
        if (HCDI_ERROR_STATS(hcdi) == NULL) {

                return;
        }

        switch (completion_reason) {
        case USB_CR_OK:
                break;
        case USB_CR_CRC:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_crc.value.ui64++;
                break;
        case USB_CR_BITSTUFFING:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_bitstuffing.value.ui64++;
                break;
        case USB_CR_DATA_TOGGLE_MM:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_data_toggle_mm.value.ui64++;
                break;
        case USB_CR_STALL:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_stall.value.ui64++;
                break;
        case USB_CR_DEV_NOT_RESP:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_dev_not_resp.value.ui64++;
                break;
        case USB_CR_PID_CHECKFAILURE:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_pid_checkfailure.value.ui64++;
                break;
        case USB_CR_UNEXP_PID:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_unexp_pid.value.ui64++;
                break;
        case USB_CR_DATA_OVERRUN:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_data_overrun.value.ui64++;
                break;
        case USB_CR_DATA_UNDERRUN:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_data_underrun.value.ui64++;
                break;
        case USB_CR_BUFFER_OVERRUN:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_buffer_overrun.value.ui64++;
                break;
        case USB_CR_BUFFER_UNDERRUN:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_buffer_underrun.value.ui64++;
                break;
        case USB_CR_TIMEOUT:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_timeout.value.ui64++;
                break;
        case USB_CR_NOT_ACCESSED:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_not_accessed.value.ui64++;
                break;
        case USB_CR_NO_RESOURCES:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_no_resources.value.ui64++;
                break;
        case USB_CR_UNSPECIFIED_ERR:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_unspecified_err.value.ui64++;
                break;
        case USB_CR_STOPPED_POLLING:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_stopped_polling.value.ui64++;
                break;
        case USB_CR_PIPE_CLOSING:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_pipe_closing.value.ui64++;
                break;
        case USB_CR_PIPE_RESET:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_pipe_reset.value.ui64++;
                break;
```



```
        case USB_CR_NOT_SUPPORTED:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_not_supported.value.ui64++;
                break;
        case USB_CR_FLUSHED:
                HCDI_ERROR_STATS_DATA(hcdi)->cc_flushed.value.ui64++;
                break;
        default:
                break;
        }
}


/*
 * Destroy the hotplug kstats structure
 */
static void
usba_hcdi_destroy_stats(usba_hcdi_t *hcdi)
{
        if (HCDI_HOTPLUG_STATS(hcdi)) {
                kstat_delete(HCDI_HOTPLUG_STATS(hcdi));
                HCDI_HOTPLUG_STATS(hcdi) = NULL;
        }

        if (HCDI_ERROR_STATS(hcdi)) {
                kstat_delete(HCDI_ERROR_STATS(hcdi));
                HCDI_ERROR_STATS(hcdi) = NULL;
        }
}


/*
 * HCD callback handling
 */
void
usba_hcdi_cb(usba_pipe_handle_data_t *ph_data,
        usb_opaque_t    req,
        usb_cr_t        completion_reason)
{

        usba_device_t           *usba_device = ph_data->p_usba_device;
        usba_hcdi_t             *hcdi = usba_hcdi_get_hcdi(
            usba_device->usb_root_hub_dip);
        usba_req_wrapper_t      *req_wrp = USBA_REQ2WRP(req);
        usb_ep_descr_t          *eptd = &ph_data->p_ep;

        mutex_enter(&ph_data->p_mutex);

#ifdef DEBUG
        mutex_enter(&ph_data->p_ph_impl->usba_ph_mutex);

        USB_DPRINTF_L4(DPRINT_MASK_HCDI, hcdi->hcdi_log_handle,
            "usba_hcdi_cb: "
            "ph_data=0x%p req=0x%p state=%d ref=%d cnt=%d cr=%d",
            (void *)ph_data, (void *)req, ph_data->p_ph_impl->usba_ph_state,
            ph_data->p_ph_impl->usba_ph_ref_count, ph_data->p_req_count,
            completion_reason);

        mutex_exit(&ph_data->p_ph_impl->usba_ph_mutex);
#endif

        /* Set the completion reason */
        switch (eptd->bmAttributes & USB_EP_ATTR_MASK) {
        case USB_EP_ATTR_CONTROL:
                ((usb_ctrl_req_t *)req)->
                    ctrl_completion_reason = completion_reason;
                break;
```

```
522         case USB_EP_ATTR_BULK:
523                 ((usb_bulk_req_t *)req)->
524                     bulk_completion_reason = completion_reason;
525                 break;
526         case USB_EP_ATTR_INTR:
527                 ((usb_intr_req_t *)req)->
528                     intr_completion_reason = completion_reason;
529                 break;
530         case USB_EP_ATTR_ISOCH:
531                 ((usb_isoc_req_t *)req)->
532                     isoc_completion_reason = completion_reason;
533                 break;
534         }

536         /*
537          * exception callbacks will still go thru a taskq thread
538          * but should occur after the soft interrupt callback
539          * By design of periodic pipes, polling will stop on any
540          * exception
541          */
542         if ((ph_data->p_spec_flag & USBA_PH_FLAG_USE_SOFT_INTR) &&
543             (completion_reason == USB_CR_OK)) {
544                 ph_data->p_soft_intr++;
545                 mutex_exit(&ph_data->p_mutex);

547                 usba_add_to_list(&hcdi->hcdi_cb_queue, &req_wrp->wr_queue);

549                 if (ddi_intr_trigger_softint(hcdi->hcdi_softint_hdl, NULL) !=
550                     DDI_SUCCESS)
551                         USB_DPRINTF_L2(DPRINT_MASK_HCDI, hcdi->hcdi_log_handle,
552                             "usba_hcdi_cb: ddi_intr_trigger_softint  failed");

554                 return;
555         }

557         /*
558          * USBA_PH_FLAG_TQ_SHARE is for bulk and intr requests,
559          * USBA_PH_FLAG_USE_SOFT_INTR is only for isoch,
560          * so there are no conflicts.
561          */
562         if (ph_data->p_spec_flag & USBA_PH_FLAG_TQ_SHARE) {
563                 int iface;

565                 mutex_exit(&ph_data->p_mutex);
566                 iface = usb_get_if_number(ph_data->p_dip);
567                 if (iface < 0) {
568                         /* we own the device, use the first taskq */
569                         iface = 0;
570                 }
571                 if (taskq_dispatch(usba_device->usb_shared_taskq[iface],
572                     hcdi_shared_cb_thread, req_wrp, TQ_NOSLEEP) ==
573                     NULL) {
574                         usba_req_exc_cb(req_wrp,
575                             USB_CR_NO_RESOURCES, USB_CB_ASYNC_REQ_FAILED);
576                 }

578                 return;
579         }

581         /* Add the callback to the pipehandles callback list */
582         usba_add_to_list(&ph_data->p_cb_queue, &req_wrp->wr_queue);

584         /* only dispatch if there is no thread running */
585         if (ph_data->p_thread_id == 0) {
586                 if (usba_async_ph_req(ph_data, hcdi_cb_thread,
587                     ph_data, USB_FLAGS_NOSLEEP) != USB_SUCCESS) {
```



```
588                         USB_DPRINTF_L2(DPRINT_MASK_HCDI, hcdi->hcdi_log_handle,
589                             "usba_hcdi_cb: taskq_dispatch failed");
590                         if (usba_rm_from_list(&ph_data->p_cb_queue,
591                             &req_wrp->wr_queue) == USB_SUCCESS) {
592                                 mutex_exit(&ph_data->p_mutex);
593                                 usba_req_exc_cb(req_wrp,
594                                     USB_CR_NO_RESOURCES,
595                                     USB_CB_ASYNC_REQ_FAILED);

597                                 return;
598                         }
599                 } else {
600                         ph_data->p_thread_id = (kthread_t *)1;
601                 }
602         }
603         mutex_exit(&ph_data->p_mutex);
604 }


607 /*
608  * thread to perform the callbacks
609  */
610 static void
611 hcdi_cb_thread(void *arg)
612 {
613         usba_pipe_handle_data_t *ph_data =
614             (usba_pipe_handle_data_t *)arg;
615         usba_ph_impl_t          *ph_impl = ph_data->p_ph_impl;
616         usba_hcdi_t             *hcdi = usba_hcdi_get_hcdi(ph_data->
617             p_usba_device->usb_root_hub_dip);
618         usba_req_wrapper_t      *req_wrp;

620         mutex_enter(&ph_data->p_mutex);
621         ASSERT(ph_data->p_thread_id == (kthread_t *)1);
622         ph_data->p_thread_id = curthread;

624         /*
625          * hold the ph_data. we can't use usba_hold_ph_data() since
626          * it will return NULL if we are closing the pipe which would
627          * then leave all requests stuck in the cb_queue
628          */
629         mutex_enter(&ph_impl->usba_ph_mutex);
630         ph_impl->usba_ph_ref_count++;

632         USB_DPRINTF_L4(DPRINT_MASK_HCDI, hcdi->hcdi_log_handle,
633             "hcdi_cb_thread: ph_data=0x%p ref=%d", (void *)ph_data,
634             ph_impl->usba_ph_ref_count);

636         mutex_exit(&ph_impl->usba_ph_mutex);

638         /*
639          * wait till soft interrupt callbacks are taken care of
640          */
641         while (ph_data->p_soft_intr) {
642                 mutex_exit(&ph_data->p_mutex);
643                 delay(1);
644                 mutex_enter(&ph_data->p_mutex);
645         }

647         while ((req_wrp = (usba_req_wrapper_t *)
648             usba_rm_first_pvt_from_list(&ph_data->p_cb_queue)) != NULL) {
649                 hcdi_do_cb(ph_data, req_wrp, hcdi);
650         }

652         ph_data->p_thread_id = 0;
653         mutex_exit(&ph_data->p_mutex);
```

```
655         USB_DPRINTF_L4(DPRINT_MASK_HCDI, hcdi->hcdi_log_handle,
656             "hcdi_cb_thread done: ph_data=0x%p", (void *)ph_data);

658         usba_release_ph_data(ph_impl);
659 }


662 static void
663 hcdi_do_cb(usba_pipe_handle_data_t *ph_data, usba_req_wrapper_t *req_wrp,
664     usba_hcdi_t *hcdi)
665 {
666         usb_cr_t                completion_reason;
667         usb_req_attrs_t         attrs = req_wrp->wr_attrs;

669         switch (req_wrp->wr_ph_data->p_ep.bmAttributes &
670             USB_EP_ATTR_MASK) {
671         case USB_EP_ATTR_CONTROL:
672                 completion_reason =
673                     USBA_WRP2CTRL_REQ(req_wrp)->ctrl_completion_reason;
674                 break;
675         case USB_EP_ATTR_INTR:
676                 completion_reason =
677                     USBA_WRP2INTR_REQ(req_wrp)->intr_completion_reason;
678                 break;
679         case USB_EP_ATTR_BULK:
680                 completion_reason =
681                     USBA_WRP2BULK_REQ(req_wrp)->bulk_completion_reason;
682                 break;
683         case USB_EP_ATTR_ISOCH:
684                 completion_reason =
685                     USBA_WRP2ISOC_REQ(req_wrp)->isoc_completion_reason;
686                 break;
687         }
688         req_wrp->wr_cr = completion_reason;

690         USB_DPRINTF_L4(DPRINT_MASK_HCDI, hcdi->hcdi_log_handle,
691             "hcdi_do_cb: wrp=0x%p cr=0x%x", (void *)req_wrp, completion_reason);

693         /*
694          * Normal callbacks:
695          */
696         if (completion_reason == USB_CR_OK) {
697                 mutex_exit(&ph_data->p_mutex);
698                 usba_req_normal_cb(req_wrp);
699                 mutex_enter(&ph_data->p_mutex);
700         } else {
701                 usb_pipe_state_t pipe_state;

703                 USB_DPRINTF_L4(DPRINT_MASK_HCDI, hcdi->hcdi_log_handle,
704                     "exception callback handling: attrs=0x%x", attrs);

706                 /*
707                  * In exception callback handling, if we were
708                  * not able to clear stall, we need to modify
709                  * pipe state. Also if auto-clearing is not set
710                  * pipe state needs to be modified.
711                  */
712                 pipe_state = usba_get_ph_state(ph_data);

714                 if (!USBA_PIPE_CLOSING(pipe_state)) {
715                         switch (completion_reason) {
716                         case USB_CR_STOPPED_POLLING:
717                                 if (pipe_state ==
718                                     USB_PIPE_STATE_ACTIVE) {
719                                         usba_pipe_new_state(ph_data,
```



```
720                                             USB_PIPE_STATE_IDLE);
721                                 }
722                                 break;
723                         case USB_CR_NOT_SUPPORTED:
724                                 usba_pipe_new_state(ph_data,
725                                     USB_PIPE_STATE_IDLE);
726                                 break;
727                         case USB_CR_PIPE_RESET:
728                         case USB_CR_FLUSHED:
729                                 break;
730                         default:
731                                 usba_pipe_new_state(ph_data,
732                                     USB_PIPE_STATE_ERROR);
733                                 break;
734                         }
735                 }

737                 pipe_state = usba_get_ph_state(ph_data);

739                 mutex_exit(&ph_data->p_mutex);
740                 if (attrs & USB_ATTRS_PIPE_RESET) {
741                         if ((completion_reason != USB_CR_PIPE_RESET) &&
742                             (pipe_state == USB_PIPE_STATE_ERROR)) {

744                                 hcdi_autoclearing(req_wrp);
745                         }
746                 }

748                 usba_req_exc_cb(req_wrp, 0, 0);
749                 mutex_enter(&ph_data->p_mutex);
750         }

752         /* Update the hcdi error kstats */
753         if (completion_reason) {
754                 mutex_enter(&hcdi->hcdi_mutex);
755                 usba_hcdi_update_error_stats(hcdi, completion_reason);
756                 mutex_exit(&hcdi->hcdi_mutex);
757         }

759         /*
760          * Once the callback is finished, release the pipe handle
761          * we start the next request first to avoid that the
762          * pipe gets closed while starting the next request
763          */
764         mutex_exit(&ph_data->p_mutex);
765         usba_start_next_req(ph_data);

767         mutex_enter(&ph_data->p_mutex);
768 }


771 /*
772  * thread to perform callbacks on the shared queue
773  */
774 static void
775 hcdi_shared_cb_thread(void *arg)
776 {
777         usba_req_wrapper_t *req_wrp = (usba_req_wrapper_t *)arg;
778         usba_pipe_handle_data_t *ph_data = req_wrp->wr_ph_data;
779         usba_ph_impl_t          *ph_impl = ph_data->p_ph_impl;
780         usba_hcdi_t             *hcdi = usba_hcdi_get_hcdi(ph_data->
781             p_usba_device->usb_root_hub_dip);
782         /*
783          * hold the ph_data. we can't use usba_hold_ph_data() since
784          * it will return NULL if we are closing the pipe which would
785          * then leave all requests stuck in the cb_queue
```

```
786          */
787         mutex_enter(&ph_impl->usba_ph_mutex);
788         ph_impl->usba_ph_ref_count++;
789
790         USB_DPRINTF_L4(DPRINT_MASK_HCDI, hcdi->hcdi_log_handle,
791             "hcdi_shared_cb_thread: ph_data=0x%p ref=%d req=0x%p",
792             (void *)ph_data, ph_impl->usba_ph_ref_count, (void *)req_wrp);
793         mutex_exit(&ph_impl->usba_ph_mutex);
794
795         /* do the callback */
796         mutex_enter(&ph_data->p_mutex);
797         hcdi_do_cb(ph_data, req_wrp, hcdi);
798         mutex_exit(&ph_data->p_mutex);
799
800         USB_DPRINTF_L4(DPRINT_MASK_HCDI, hcdi->hcdi_log_handle,
801             "hcdi_cb_thread done: ph_data=0x%p", (void *)ph_data);
802
803         usba_release_ph_data(ph_impl);
804 }
805
806
807 /*
808  * soft interrupt handler
809  */
810 /*ARGSUSED*/
811 static uint_t
812 hcdi_soft_intr(caddr_t arg1, caddr_t arg2)
813 {
814         usba_hcdi_t             *hcdi = (void *)arg1;
815         usba_req_wrapper_t      *req_wrp;
816         int                     count = 0;
817
818         while ((req_wrp = (usba_req_wrapper_t *)
819             usba_rm_first_pvt_from_list(&hcdi->hcdi_cb_queue)) != NULL) {
820                 usba_pipe_handle_data_t *ph_data = req_wrp->wr_ph_data;
821                 usba_ph_impl_t          *ph_impl = ph_data->p_ph_impl;
822
823                 /* hold the pipe */
824                 mutex_enter(&ph_impl->usba_ph_mutex);
825                 ph_impl->usba_ph_ref_count++;
826                 mutex_exit(&ph_impl->usba_ph_mutex);
827
828                 /* do the callback */
829                 usba_req_normal_cb(req_wrp);
830
831                 /* decrement the soft interrupt count */
832                 mutex_enter(&ph_data->p_mutex);
833                 ph_data->p_soft_intr--;
834                 mutex_exit(&ph_data->p_mutex);
835
836                 /* release the pipe */
837                 mutex_enter(&ph_impl->usba_ph_mutex);
838                 ph_impl->usba_ph_ref_count--;
839                 mutex_exit(&ph_impl->usba_ph_mutex);
840
841                 count++;
842         }
843
844         return (count == 0 ? DDI_INTR_UNCLAIMED : DDI_INTR_CLAIMED);
845 }
846
847
848 /*
849  * hcdi_autoclearing:
850  *      This function is called under the taskq context. It
851  *      resets the pipe, and clears the stall, if necessary
```



```
852  */
853 static void
854 hcdi_autoclearing(usba_req_wrapper_t *req_wrp)
855 {
856         usb_cr_t                cr = req_wrp->wr_cr;
857         usb_pipe_handle_t       pipe_handle, def_pipe_handle;
858         usb_cr_t                completion_reason;
859         usb_cb_flags_t          cb_flags;
860         int                     rval;
861         usba_device_t           *usba_device =
862             req_wrp->wr_ph_data->p_usba_device;
863         usba_hcdi_t             *hcdi = usba_hcdi_get_hcdi(
864             usba_device->usb_root_hub_dip);
865         usb_req_attrs_t         attrs = req_wrp->wr_attrs;
866
867         USB_DPRINTF_L4(DPRINT_MASK_HCDI, hcdi->hcdi_log_handle,
868             "hcdi_autoclearing: wrp=0x%p", (void *)req_wrp);
869
870         pipe_handle = usba_get_pipe_handle(req_wrp->wr_ph_data);
871         def_pipe_handle = usba_get_dflt_pipe_handle(req_wrp->wr_ph_data->p_dip);
872
873         /*
874          * first reset the pipe synchronously
875          */
876         if ((attrs & USB_ATTRS_PIPE_RESET) == USB_ATTRS_PIPE_RESET) {
877                 usba_pipe_clear(pipe_handle);
878                 usba_req_set_cb_flags(req_wrp, USB_CB_RESET_PIPE);
879         }
880
881         ASSERT(def_pipe_handle);
882
883         /* Do not clear if this request was a usb_get_status request */
884         if ((pipe_handle == def_pipe_handle) &&
885             (USBA_WRP2CTRL_REQ(req_wrp)->ctrl_bRequest ==
886             USB_REQ_GET_STATUS)) {
887                 USB_DPRINTF_L2(DPRINT_MASK_USBAI, hcdi->hcdi_log_handle,
888                     "hcdi_autoclearing: usb_get_status failed, no clearing");
889
890         /* if default pipe and stall no auto clearing */
891         } else if ((pipe_handle == def_pipe_handle) && (cr == USB_CR_STALL)) {
892                 USB_DPRINTF_L2(DPRINT_MASK_USBAI, hcdi->hcdi_log_handle,
893                     "hcdi_autoclearing: default pipe stalled, no clearing");
894
895                 usba_req_set_cb_flags(req_wrp, USB_CB_PROTOCOL_STALL);
896
897         /* else do auto clearing */
898         } else if (((attrs & USB_ATTRS_AUTOCLEARING) ==
899             USB_ATTRS_AUTOCLEARING) && (cr == USB_CR_STALL)) {
900                 ushort_t status = 0;
901
902                 rval = usb_get_status(req_wrp->wr_dip, def_pipe_handle,
903                     USB_DEV_REQ_DEV_TO_HOST | USB_DEV_REQ_RCPT_EP,
904                     req_wrp->wr_ph_data->p_ep.bEndpointAddress,
905                     &status, USB_FLAGS_SLEEP);
906                 if (rval != USB_SUCCESS) {
907                         USB_DPRINTF_L2(DPRINT_MASK_USBAI, hcdi->hcdi_log_handle,
908                             "get status (STALL) failed: rval=%d", rval);
909
910                         usba_pipe_clear(def_pipe_handle);
911                 }
912
913                 if ((rval != USB_SUCCESS) ||
914                     (status & USB_EP_HALT_STATUS)) {
915                         usba_req_set_cb_flags(req_wrp, USB_CB_FUNCTIONAL_STALL);
916
917                         if ((rval = usb_pipe_sync_ctrl_xfer(
```

```
                            req_wrp->wr_dip, def_pipe_handle,
                            USB_DEV_REQ_HOST_TO_DEV |
                            USB_DEV_REQ_RCPT_EP,
                            USB_REQ_CLEAR_FEATURE,
                            0,
                            req_wrp->wr_ph_data->p_ep.bEndpointAddress,
                            0,
                            NULL, 0,
                            &completion_reason,
                            &cb_flags, USB_FLAGS_SLEEP)) != USB_SUCCESS) {
                                USB_DPRINTF_L2(DPRINT_MASK_USBAI,
                                    hcdi->hcdi_log_handle,
                                    "auto clearing (STALL) failed: "
                                    "rval=%d, cr=0x%x cb=0x%x",
                                    rval, completion_reason, cb_flags);

                                usba_pipe_clear(def_pipe_handle);
                        } else {
                                usba_req_set_cb_flags(req_wrp,
                                    USB_CB_STALL_CLEARED);
                        }
                } else {
                        usba_req_set_cb_flags(req_wrp, USB_CB_PROTOCOL_STALL);
                }
        }
}


/*
 * usba_hcdi_get_req_private:
 *      This function is used to get the HCD private field
 *      maintained by USBA. HCD calls this function.
 *
 * Arguments:
 *      req             - pointer to usb_*_req_t
 *
 * Return Values:
 *      wr_hcd_private field from wrapper
 */
usb_opaque_t
usba_hcdi_get_req_private(usb_opaque_t req)
{
        usba_req_wrapper_t *wrp = USBA_REQ2WRP(req);

        return (wrp->wr_hcd_private);
}


/*
 * usba_hcdi_set_req_private:
 *      This function is used to set the HCD private field
 *      maintained by USBA. HCD calls this function.
 *
 * Arguments:
 *      req             - pointer to usb_*_req_t
 *      hcd_private     - wr_hcd_private field from wrapper
 */
void
usba_hcdi_set_req_private(usb_opaque_t req,
                        usb_opaque_t    hcd_private)
{
        usba_req_wrapper_t *wrp = USBA_REQ2WRP(req);

        wrp->wr_hcd_private = hcd_private;
}
```



```
/* get data toggle information for this endpoint */
uchar_t
usba_hcdi_get_data_toggle(usba_device_t *usba_device, uint8_t ep_addr)
{
        uchar_t         toggle;
        usba_ph_impl_t  *ph_impl;
        int             ep_index;

        ep_index = usb_get_ep_index(ep_addr);
        mutex_enter(&usba_device->usb_mutex);
        ph_impl = &usba_device->usb_ph_list[ep_index];
        mutex_enter(&ph_impl->usba_ph_mutex);
        toggle = (uchar_t)(ph_impl->usba_ph_flags & USBA_PH_DATA_TOGGLE);
        mutex_exit(&ph_impl->usba_ph_mutex);
        mutex_exit(&usba_device->usb_mutex);

        return (toggle);
}


/* set data toggle information for this endpoint */
void
usba_hcdi_set_data_toggle(usba_device_t *usba_device, uint8_t ep_addr,
    uchar_t toggle)
{
        usba_ph_impl_t  *ph_impl;
        int             ep_index;

        ep_index = usb_get_ep_index(ep_addr);
        mutex_enter(&usba_device->usb_mutex);
        ph_impl = &usba_device->usb_ph_list[ep_index];
        mutex_enter(&ph_impl->usba_ph_mutex);
        ph_impl->usba_ph_flags &= ~USBA_PH_DATA_TOGGLE;
        ph_impl->usba_ph_flags |= (USBA_PH_DATA_TOGGLE & toggle);
        mutex_exit(&ph_impl->usba_ph_mutex);
        mutex_exit(&usba_device->usb_mutex);
}


/* get pipe_handle_impl ptr for this ep */
usba_pipe_handle_data_t *
usba_hcdi_get_ph_data(usba_device_t *usba_device, uint8_t ep_addr)
{
        return (usba_device->usb_ph_list[usb_get_ep_index(ep_addr)].
            usba_ph_data);
}
```