

UDI Tutorial & Driver Walk-Through

Part 2

<http://www.sco.com/forum1999/conference/developfast/f10>

Kurt Gollhardt
SCO Core OS Architect
E-mail: kdg@sco.com



Overview

- Key Concepts
- Description of Sample Driver
- **Walk-Through**
- **Additional Examples**
- **Hints & Tips**



Walk-Through

udi_cmos.c lines 243-248

- **Primary region initialization**

```
243  static const udi_primary_init_t
      udi_cmos_primary_init_info = {
244      &cmos_mgmt_ops,
245      0,    /* mgmt_scratch_size */
246      0,    /* no children, so no enumeration */
247      sizeof(cmos_region_data_t)
248  };
```



Walk-Through

udi_cmos.c lines 255-285

- Channel ops initialization

```
255     static const udi_ops_init_t
           udi_cmos_ops_init_list[] = {
256         {
257             GIO_OPS_IDX,
258             CMOS_GIO_META,
260             UDI_GIO_PROVIDER_OPS_NUM,
261             0,          /* no channel context */
262             (udi_ops_vector_t *)
                &cmos_gio_provider_ops
263         }, ...
```



Walk-Through

udi_cmos.c lines 292-346

- **Control block initialization**

```
292     static const udi_cb_init_t
           udi_cmos_cb_init_list[] = {
293         {
294             GIO_BIND_CB_IDX,
295             CMOS_GIO_META,
296             UDI_GIO_BIND_CB_NUM,
297             GIO_BIND_SCRATCH,
298             0,          /* inline size */
299             NULL        /* inline layout */
300         }, ...
```



Walk-Through

udi_cmos.c lines 348-355

- **Driver initialization structure**

```
348     const udi_init_t udi_init_info = {
349         &udi_cmos_primary_init_info,
350         NULL,          /* secondary_init_list */
351         udi_cmos_ops_init_list,
352         udi_cmos_cb_init_list,
353         NULL,          /* gcb_init_list */
354         NULL           /* cb_select_list */
355     };
```

- **udi_init_info is only well-known global name**



Walk-Through

udi_cmos.c lines 365-381

- Handle channel events from parent (bridge)

```
365  static void
366  cmos_parent_channel_event(udi_channel_event_cb_t *channel_event_cb)
367  {
368      switch (channel_event_cb->event) {
369          case UDI_CHANNEL_BOUNDED:
370              /* Allocate a bus bind control block. */
371              udi_cb_alloc(cmos_bind_to_parent_1,
372                          UDI_GCB(channel_event_cb),
373                          BUS_BIND_CB_IDX,
374                          channel_event_cb->gcb.channel);
375              break;
376          default:
377              udi_channel_event_complete(channel_event_cb, UDI_OK);
378      }
379  }
380
381 }
```



Walk-Through

udi_cmos.c lines 384-396

- Continue bind sequence after allocation

```
384  cmos_bind_to_parent_1(  
385      udi_cb_t *gcb,  
386      udi_cb_t *new_cb)  
387  {  
388      udi_bus_bind_cb_t *bus_bind_cb =  
389          UDI_MCB(new_cb, udi_bus_bind_cb_t);  
390  
391      /* Keep a link back to the channel event CB for the ack. */  
392      bus_bind_cb->gcb.initiator_context = gcb;  
393  
394      /* Bind to the parent bus bridge driver. */  
395      udi_bus_bind_req(bus_bind_cb);  
396  }
```



Walk-Through

udi_cmos.c lines 400-429

- Received ack from bridge; map PIO device

```
400 static void
401 cmos_bus_bind_ack(
402     udi_bus_bind_cb_t *bus_bind_cb,
403     udi_ubit8_t preferred_endianness,
404     udi_status_t status)
405 {
406     cmos_region_data_t *rdata = bus_bind_cb->gcb.context;
418     rdata->bus_bind_cb = bus_bind_cb;
424     udi_pio_map(cmos_bus_bind_ack_1, UDI_GCB(bus_bind_cb),
425                 CMOS_REGSET, CMOS_BASE, CMOS_LENGTH,
426                 cmos_trans_read,
427                 sizeof cmos_trans_read /
428                 sizeof(udi_pio_trans_t),
429                 UDI_PIO_STRICTORDER, CMOS_PACE);
429 }
```



Walk-Through

udi_cmos.c lines 434-448

- **Map second PIO handle (2 trans lists)**

```
434  cmos_bus_bind_ack_1(  
435      udi_cb_t *gcb,  
436      udi_pio_handle_t new_pio_handle)  
437  {  
438      cmos_region_data_t *rdata = gcb->context;  
441      rdata->trans_read = new_pio_handle;  
442  
443      udi_pio_map(cmos_bus_bind_ack_2, gcb,  
444                  CMOS_REGSET, CMOS_BASE, CMOS_LENGTH,  
445                  cmos_trans_write,  
446                  sizeof cmos_trans_write /  
                      sizeof(udi_pio_trans_t),  
447                  UDI_PIO_STRICTORDER, CMOS_PACE);  
448  }
```



Walk-Through

udi_cmos.c lines 451-468

- **Bind sequence complete**

```
451  cmos_bus_bind_ack_2(  
452      udi_cb_t *gcb,  
453      udi_pio_handle_t new_pio_handle)  
454  {  
455      cmos_region_data_t *rdata = gcb->context;  
456      udi_channel_event_cb_t *channel_event_cb =  
457          gcb->initiator_context;  
458  
459      /* Save the PIO handle for later use. */  
460      rdata->trans_write = new_pio_handle;  
465  
466      /* Let the MA know we've completed binding. */  
467      udi_channel_event_complete(channel_event_cb, UDI_OK);  
468  }
```



Walk-Through

udi_cmos.c lines 470-480

- **Child bind and unbind**
 - This driver has no per-child state

```
470 static void
471 cmos_gio_bind_req(udi_gio_bind_cb_t *gio_bind_cb)
472 {
473     udi_gio_bind_ack(gio_bind_cb, CMOS_DEVSZ, 0, UDI_OK);
474 }
475
476 static void
477 cmos_gio_unbind_req(udi_gio_bind_cb_t *gio_bind_cb)
478 {
479     udi_gio_unbind_ack(gio_bind_cb);
480 }
```



Walk-Through

udi_cmos.c lines 492-508

- Transfer request operation - READ case

```
492  cmos_gio_xfer_req(udi_gio_xfer_cb_t *gio_xfer_cb)
493  {
494      udi_gio_rw_params_t *rw_params = gio_xfer_cb->tr_params;
495
501      switch (gio_xfer_cb->op) {
502      case UDI_GIO_OP_READ:
503          udi_assert(rw_params->offset_hi == 0 &&
504                  rw_params->offset_lo < CMOS_DEVSZ &&
505                  gio_xfer_cb->data_buf->buf_size <
506                  CMOS_DEVSZ - rw_params->offset_lo);
507      cmos_do_read(gio_xfer_cb, rw_params->offset_lo);
508      break;
```



Walk-Through

udi_cmos.c lines 509-519

- Transfer request operation - WRITE case

```
509         case UDI_GIO_OP_WRITE:
510             udi_assert(rw_params->offset_hi == 0 &&
511                 rw_params->offset_lo < CMOS_DEVSZ &&
512                 gio_xfer_cb->data_buf->buf_size <
513                 CMOS_DEVSZ - rw_params->offset_lo);
514             cmos_do_write(gio_xfer_cb, rw_params->offset_lo);
515             break;
```

- Transfer request operation - error case

```
516         default:
517             udi_gio_xfer_nak(gio_xfer_cb, UDI_STAT_NOT_UNDERSTOOD);
518         }
519     }
```



Walk-Through

udi_cmos.c lines 525-541

- **Initiate PIO transactions for READ request**

```
525  cmos_do_read(udi_gio_xfer_cb_t *gio_xfer_cb, udi_ubit8_t addr)
526  {
527      cmos_region_data_t *rdata = gio_xfer_cb->gcb.context;
528      cmos_gio_xfer_scratch_t *gio_xfer_scratch =
529          gio_xfer_cb->gcb.scratch;
530
531
532
533
534
535      gio_xfer_scratch->addr = addr;
536      gio_xfer_scratch->count = gio_xfer_cb->data_buf->buf_size;
537
538
539
540
541      udi_pio_trans(cmos_do_read_1, gcb,
542                  rdata->trans_read,
543                  new_buf, NULL);
544  }
```



Walk-Through

udi_cmos.c lines 544-560

- **Complete READ request**

```
544  cmos_do_read_1(  
545      udi_cb_t *gcb,  
546      udi_buf_t *new_buf,  
547      udi_status_t status,  
548      udi_ubit16_t result)  
549  {  
550      udi_gio_xfer_cb_t *gio_xfer_cb =  
551          UDI_MCB(gcb, udi_gio_xfer_cb_t);  
552  
553      /* udi_pio_trans may create a new buffer. */  
554      gio_xfer_cb->data_buf = new_buf;  
555  
556      if (status == UDI_OK)  
557          udi_gio_xfer_ack(gio_xfer_cb);  
558      else  
559          udi_gio_xfer_nak(gio_xfer_cb, status);  
560  }
```



Walk-Through

udi_cmos.c lines 566-613

- **WRITE case identical to READ except:**
 - Uses trans_write instead of trans_read
 - Special case for CMOS driver:

```
572         /*
573         * The first CMOS_RDONLY_SZ bytes of this device are not
574         * allowed to be written through this driver. Fail any
575         * attempt to write to these bytes.
576         */
577         if (addr < CMOS_RDONLY_SZ) {
578             udi_gio_xfer_nak(gio_xfer_cb,
579                             UDI_STAT_MISTAKEN_IDENTITY);
580             return;
581         }
```



Walk-Through

udi_cmos.c lines 615-636

- **Device Management - unbind from parent**

```
616     cmos_devmgmt_req(  
617         udi_mgmt_cb_t *cb,  
618         udi_ubit8_t mgmt_op, udi_ubit8_t parent_id)  
620     {  
621         cmos_region_data_t *rdata = cb->gcb.context;  
623         switch (mgmt_op) {  
624             case UDI_DMGMGT_UNBIND:  
628                 /* Keep a link back to this CB for use in the ack */  
629                 rdata->bus_bind_cb->gcb.initiator_context = cb;  
631                 /* Do the metalanguage-specific unbind. */  
632                 udi_bus_unbind_req(rdata->bus_bind_cb);  
633             default:  
634                 udi_devmgmt_ack(cb, 0, UDI_OK);  
635         }  
636     }
```



Walk-Through

udi_cmos.c lines 639-646

- **Complete parent unbind sequence**

```
639  cmos_bus_unbind_ack(udi_bus_bind_cb_t *bus_bind_cb)
640  {
641      udi_mgmt_cb_t *cb = bus_bind_cb->gcb.initiator_context;
642
643      udi_cb_free(UDI_GCB(bus_bind_cb));
644
645      udi_devmgmt_ack(cb, 0, UDI_OK);
646  }
```



Walk-Through

udi_cmos.c lines 384-396

- **Final cleanup**

- Called after all children/parents unbound

```
649  cmos_final_cleanup_req(udi_mgmt_cb_t *cb)
650  {
651      /*
652       * We have nothing to free that wasn't already freed by
653       * unbinding children and parents.
654       */
655      udi_final_cleanup_ack(cb);
656  }
```



PIO Transaction List for READ

udi_cmos.c lines 174-185

```
174      /* R0 <- SCRATCH_ADDR {offset into scratch of address} */
175      { UDI_PIO_LOAD_IMM+UDI_PIO_R0, UDI_PIO_2BYTE, SCRATCH_ADDR },
176      /* R1 <- address */
177      { UDI_PIO_LOAD+UDI_PIO_SCRATCH+UDI_PIO_R0,
178          UDI_PIO_1BYTE, UDI_PIO_R1 },
179      /* R0 <- SCRATCH_COUNT {offset into scratch of count} */
180      { UDI_PIO_LOAD_IMM+UDI_PIO_R0, UDI_PIO_2BYTE, SCRATCH_COUNT },
181      /* R2 <- count */
182      { UDI_PIO_LOAD+UDI_PIO_SCRATCH+UDI_PIO_R0,
183          UDI_PIO_1BYTE, UDI_PIO_R2 },
184      /* R0 <- 0 {current buffer offset} */
185      { UDI_PIO_LOAD_IMM+UDI_PIO_R0, UDI_PIO_2BYTE, 0 },
```



PIO Transaction List for READ

udi_cmos.c lines 186-203

```
186      /* begin main loop */
187      { UDI_PIO_LABEL, 0, 1 },
188      /* output address from R1 to address register */
189      { UDI_PIO_OUT+UDI_PIO_DIRECT+UDI_PIO_R1,
190          UDI_PIO_1BYTE, CMOS_ADDR },
191      /* input value from data register into next buffer location */
192      { UDI_PIO_IN+UDI_PIO_BUF+UDI_PIO_R0,
193          UDI_PIO_1BYTE, CMOS_DATA },
194      /* decrement count (in R2) */
195      { UDI_PIO_ADD_IMM+UDI_PIO_R2, UDI_PIO_1BYTE, (udi_ubit8_t)-1 },
196      /* if count is zero, we're done */
197      { UDI_PIO_CSKIP+UDI_PIO_R2, UDI_PIO_1BYTE, UDI_PIO_NZ },
198      { UDI_PIO_END, UDI_PIO_2BYTE, 0 },
199      /* increment address and buffer offset */
200      { UDI_PIO_ADD_IMM+UDI_PIO_R1, UDI_PIO_1BYTE, 1 },
201      { UDI_PIO_ADD_IMM+UDI_PIO_R0, UDI_PIO_1BYTE, 1 },
202      /* go back to main loop */
203      { UDI_PIO_BRANCH, 0, 1 }
```



Additional Examples

Memory Allocation

- **Asynchronous service call**

```
udi_mem_alloc(callback, gcb, size, UDI_MEM_NOZERO);
```

- **Callback**

```
static void callback(  
    udi_cb_t *gcb,  
    void *new_mem )  
{  
    /* continue */  
}
```



Additional Examples

Memory Allocation - Limits

- **Size > limits.max_safe_alloc**
 - Driver must cancel allocation if it takes too long
 - » Start timer with `udi_timer_start`
 - » Cancel allocation with `udi_cancel`
 - » On success, cancel timer w/ `udi_timer_cancel`
- **Size <= UDI_MIN_ALLOC_LIMIT**
 - Don't need runtime test



Additional Examples

Timers

- **Timer interval stored as `udi_time_t`**

– Seconds, nanoseconds

```
udi_time_t interval = { 0, 1000000 };  
udi_timer_start(callback, gcb, interval);
```

- **Timer callback**

```
static void callback(gcb);
```

- **Cancelling a timer**

```
udi_timer_cancel(gcb);
```



Hints & Tips

- For debugging, try `udi_debug_break()`.
- To print debug info, use `udi_trace_write()`.
- Don't forget to cancel pending timers during unbinding.
- Keep pools of control blocks and buffers instead of allocating new ones for each request.

