```
 2 ____________________________________________________________________________
34 /*
35 Helpful Linux commands
36         dmesg   - retrieve printk() messages.  -c flag clears the buffer.
37         modinfo - retrieve information regarding the compiled .ko module
38         insmod  - install a module
39         lsmod   - list modules loaded in the kernel
40         rmmod   - remove module
41         modprobe - insmod plus dependencies
42 */
43
44 /*****************************************************************************/
45 /*****************************************************************************/
46 /*****************************************************************************/
47 /*    < < < INCLUDES  > > >                                                  */
48 /*****************************************************************************/
49
50
51
52 /*****************************************************************************/
53 #include <linux/version.h>      /* LINUX_VERSION_CODE, KERNEL_VERSION   */
54 #include <linux/init.h>         /* module_init, module_exit             */
55 #include <linux/kernel.h>
56 #include <linux/module.h>       /* for modules                          */
57 #include <linux/moduleparam.h>
58 #include <linux/cdev.h>
59 #include <linux/fs.h>           /* file_operations                      */
60 #include <linux/uaccess.h>      /* copy_(to,from)_user and access_ok()  */
61 #include <linux/device.h>       /* sysfs class                          */
62 //#include <linux/slab.h>       /* kmalloc                              */
63 #include <linux/errno.h>        /* return error codes                   */
64
65 #if(0)
66 #include "../aul_tree/shared/compiler.h"
67 #include "../aul_tree/shared/device.h"
68 #include "../aul_tree/shared/status.h"
69 #include "../aul_tree/shared/cmd.h"
70 #include "../aul_tree/shared/module_macros.h"
71 #include "../aul_tree/str_safe/str_safe.h"
72 #include "../aul_tree/io/io_shared.h"
```

```
 73 #endif
 74
 75 #include "../str_safe/str_safe.h"
 76 #include "../shared/cmd_macros.h"
 77 #include "../shared/cmd.h"
 78 #include "../shared/device_list.h"   /* struct device_list */
 79 #include "../shared/device.h"        /* DEVICE_QTY         */
 80
 81 #include "../shared/status.h"
 82 #include "../io/io_shared.h"
 83 #include "../io/io_private.h"  /* IO_COPY_TO/FROM_USER() */
 84
 85 #include "../io/io.h"
 86 #include "../shared/ts_svn.h"     /* extract subversion date code */
 87
 88 #include "../devices/dev_avail_module.h"  /* DEVICE_STX104, DEVICE_TRACERE */
 89
 90 /*****************************************************************************/
 91 #define AULDRV_MODULE_REV  "$Date: 2015-02-05 06:26:19 -0600 (Thu, 05 Feb 2015) $"
 92 /*****************************************************************************/
 93 #ifndef MODULE_AULDRV
 94 # define MODULE_AULDRV      7 /* module ID number */
 95 # define MODULE_AULDRV_NAME DRIVER_NAME    /* must be 'auldrv' */
 96 #endif /* MODULE_AULDRV */
 97
 98 #define DBG_DRV_NAME  MODULE_AULDRV_NAME
 99
100 /*****************************************************************************/
101 /*****************************************************************************/
102 /*****************************************************************************/
103 /*              < < < DEBUG PARAMS > > >                                     */
104 /*****************************************************************************/
105
106 /*****************************************************************************/
107 /* if DEBUG then allow AULDRV_DEBUG */
108 #define AULDRV_DEBUG          _DEBUG
109 /*****************************************************************************/
110 /* comment out the line below if you wish to remove
111    debug output from this module                                             */
112
113 #define AULDRV_DEBUG_LOG   DEBUG_OUT
```

```
114
115
116
117
118 /*****************************************************************************/
119 /* DO NOT USE */
120 /* #define AULDRV_DEVICE_DYNAMIC 1 */
121 /*****************************************************************************/
122
123 enum { AULDRV_DRIVER_MINOR_NUMBER = 0    };
124 enum { AULDRV_DRIVER_MINOR_INVALID = 100 }; /* TODO: use max unsigned value */
125
126 /*****************************************************************************/
127 /* TODO: merge with debug.c and debug.h */
128 #undef PDEBUG             /* undef it, just in case */
129 #if ( defined( __KERNEL__ ) && defined( _DEBUG ) )
130   /* This one if debugging is on, and kernel space */
131 //#  include <linux/kernel.h>
132 #  ifdef DRIVER_NAME
133 #    define PDEBUG(fmt, args...) printk( KERN_DEBUG DBG_DRV_NAME " " fmt, ## args )
134 #    pragma message "PDEBUG: __KERNEL__ and DRIVER_NAME=" DRIVER_NAME "."
135 #  else
136 #    define PDEBUG(fmt, args...) printk( KERN_DEBUG "" fmt, ## args )
137 #    pragma message "PDEBUG: __KERNEL__ and no DRIVER_NAME"
138 #  endif
139 #elif ( DEBUG_APPLICATION )
140    /* This one for user space */
141 #  include <stdio.h>
142 #  ifdef  DRIVER_NAME
143 #    define PDEBUG(fmt, args...) fprintf( stderr, DBG_DRV_NAME fmt, ## args )
144 #    pragma message "PDEBUG: no __KERNEL__, DRIVER_NAME=" DRIVER_NAME "."
145 #  else
146 #    define PDEBUG(fmt, args...) fprintf( stderr, fmt, ## args )
147 #    pragma message "PDEBUG: DEBUG_APPLICATION and no DRIVER_NAME"
148 #  endif
149 #else
150 #  define PDEBUG( fmt, args... ) /* not debugging: nothing */
151 #  pragma message "PDEBUG: no debug reporting."
152 #endif
153 /*****************************************************************************/
154
```

```
155
156 /*****************************************************************************/
157 /*****************************************************************************/
158 /*****************************************************************************/
159 /*   < < < MODULE PARAMETERS > > >                                           */
160 /*****************************************************************************/
161 #if !defined( MODULE_MACROS_H_ )
162 # pragma message ( "aul_tree: auldrv.c  MODULE_MACRO_H_ not defined" )
163 /*****************************************************************************/
164 typedef int  ( * module_fnc     )( void );
165 /*****************************************************************************/
166 struct module_definition
167 {
168 //  const SYS_TYPE      type;
169
170     module_fnc      initialize;
171     module_fnc      terminate;
172     /* Purpose of revision information:
173             1. Module management.
174             2. Determine if application and module status/commands are in sync.
175     */
176     const char *    name;
177     const char *    svn_module_date;
178     const char *    svn_command_date;
179     const char *    svn_status_date;
180 };
181 typedef struct module_definition auldrv_module_definition_type;
182 #endif
183
184 /*****************************************************************************/
185 #define IO_DEFINITION_ACRONYM_SIZE  16
186 /*****************************************************************************/
187 #if !defined( IO_SHARED_H__ )
188 # pragma message ( "aul_tree: auldrv.c  IO_SHARED_H__ not defined" )
189 struct io_definition
190 {
191     /* physical_id
192         The physical id is both the logical_id along with the instance.
193     */
194     IOC_T               physical_id;
195     /* minor number
```

```
196         Same as board number within driver level.
197     */
198     size_t          minor_number;
199     size_t          region_offset;
200     size_t          region_bytes;
201     size_t          region_restrict_8bit;  /* this is a bus width restriction */
202     size_t          interrupt_number;
203     /* acronym
204         Same name given to file nodes.  For example, if we have two tracere boards,
205         then the acronym will be "tracere0" and "tracere1" where the minor number
206         is appended to the generic acronym.  If only a generic acronym is available
207         and sent to the driver to retrieve information, it will pull the information
208         for the device that is not active.  For example, if we send "tracere" and
209       we have "tracer0" and "tracer1", and "tracer0" has already been open, it will
210       then simply return the information for "tracer1" and update the acronym to
211         support it.  If we had sent "tracer0" and it was open and we try to open with
212         it, it will return an error (i.e. IO_Definition_Get() ).
213
214         <acronym><minor_number>
215      */
216     char                acronym[IO_DEFINITION_ACRONYM_SIZE];
217 };
218 typedef struct io_definition io_definition_type;
219 #endif
220 /*  NOTE: in this particular case "region" means the same as "device"       */
221 /*************************************************************************/
222 /*#define AULDRV_REGION_QTY     2*/
223 #define AULDRV_REGION_QTY       DEVICE_QTY
224 /*************************************************************************/
225 struct auldrv_region
226 {
227     int           io_descriptor;
228     size_t        open_and_active;
229     size_t        offset;
230     size_t        bytes;
231     size_t        restrict_8bits;
232     char *        name;
233 };
234 /*************************************************************************/
235 struct auldrv_driver_params
236 {
```

```
237     unsigned                    major;  /* major number             */
238     unsigned                    minor;  /* minor number             */
239     dev_t                       number; /* device number            */
240     char *                      name;   /* name of this driver      */
241     struct cdev                 cdev;   /* driver character device  */
242     unsigned                    minor_count; /* quantity of minor numbers */
243     int                         open_and_active;
244     int                         error_code_last;
245     struct module_definition    module_aul; /* our driver definition of a module */
246     struct class *              class;  /* driver class             */
247 };
248
249 /*****************************************************************************/
250 struct auldrv_device_params
251 {
252     unsigned                major;  /* major number             */
253     unsigned                minor;  /* minor number             */
254     dev_t                   number; /* device number            */
255     char *                  name;   /* name of this device      */
256     struct cdev             cdev;   /* driver character device  */
257     int                     open_and_active;
258     int                     error_code_last;
259     int                     io_descriptor;
260     struct io_definition    region;
261 };
262 /*****************************************************************************/
263 struct auldrv_dataset
264 {
265     struct auldrv_driver_params     driver;
266     unsigned                        map_minor_to_device[AULDRV_REGION_QTY + 1 /* driver/controller */];
267     unsigned                        device_count;
268     struct auldrv_device_params     device[AULDRV_REGION_QTY];
269 };
270 typedef struct auldrv_dataset_type;
271
272 /*****************************************************************************/
273
274
275 /*****************************************************************************/
276 /*****************************************************************************/
277 /*                 < < < VARIABLES > > >                                     */
```

```
278 /*****************************************************************************/
279
280 static const char * auldrv_module_name = MODULE_AULDRV_NAME;
281 /* Used to report back the name of this kernel module */
282 static const char auldrv_ioctl_module_name[] = "mod_"  MODULE_AULDRV_NAME;
283 static struct auldrv_dataset auldrv = { { 0 } };
284
285
286 /*****************************************************************************/
287 /*****************************************************************************/
288 /*****************************************************************************/
289 /*    < < < MODULE PARAMETERS > > >                                          */
290 /*****************************************************************************/
291
292 /*****************************************************************************
293 Module parameters passed in by insmod or modprobe.
294
295 TODO:
296 1. improve flexibility.  For now, these MUST be defined.
297 2. move all parameters into a data structure for easy modifications.
298
299 Note: These parameters will take priority over any library parameters.
300
301       What are these parameters?  They describe the hardware base address,
302       I/O space to allocate, the device name and whether or not to restrict
303       I/O transactions to byte width.   These parameters describe an
304       ISA bus interface.
305
306     offset   - sets the base address of each region.
307     bytes    - sets the contiguous bytes at a given I/O base address.
308     restrict - restricts data transactions to 8-bit bus width.
309     acronym  - name of the region (or device).
310 */
311
312 static int auldrv_requested_driver_major_number = 0;
313 module_param_named( major, auldrv_requested_driver_major_number, int, S_IRUGO );
314 MODULE_PARM_DESC( major, "This driver ID. Dynamic assignment when major=0 (default)." );
315
316 /* NOTE: double {} is a GCC work around */
317 static unsigned int auldrv_requested_region_offset[AULDRV_REGION_QTY] = { { 0 } };
318 static int auldrv_requested_region_offset_count = 0;
```

```
319 module_param_array_named(offset, auldrv_requested_region_offset, uint, &auldrv_requested_region_offset_count, S_IRUGO );
320 MODULE_PARM_DESC( offset, "region offset array. Ex: offset=0x300 or offset=0x300,0x310" );
321
322 static unsigned int auldrv_requested_region_bytes[AULDRV_REGION_QTY] = {  0, };
323 static int auldrv_requested_region_bytes_count = 0;
324 module_param_array_named( bytes, auldrv_requested_region_bytes, uint, &auldrv_requested_region_bytes_count, S_IRUGO );
325 MODULE_PARM_DESC( bytes, "region bytes array. Ex: bytes=0x10 or bytes=0x10,0x10" );
326
327 static unsigned int auldrv_requested_region_restrict_8bit[AULDRV_REGION_QTY] = { { 0 } };
328 static int auldrv_requested_region_restrict_8bit_count = 0;
329 module_param_array_named( restrict, auldrv_requested_region_restrict_8bit, uint, &auldrv_requested_region_restrict_8bit_count,
    S_IRUGO );
330 MODULE_PARM_DESC( bit8, "region restrict 8-bits. Ex: restrict=0 or restrict=0,1" );
331
332 static char * auldrv_requested_region_acronym[AULDRV_REGION_QTY] = { NULL, };
333 static int auldrv_requested_region_acronym_count = 0;
334 module_param_array_named( acronym, auldrv_requested_region_acronym, charp, &auldrv_requested_region_acronym_count, S_IRUGO );
335 MODULE_PARM_DESC( acronym, "region name array. Ex: acronym=\"stx104\" or acronym=\"stx104\",\"tracer\"" );
336
337 static char * auldrv_requested_region_type[AULDRV_REGION_QTY] = { NULL, };
338 static int auldrv_requested_region_type_count = 0;
339 module_param_array_named( type, auldrv_requested_region_type, charp, &auldrv_requested_region_type_count, S_IRUGO );
340 MODULE_PARM_DESC( type, "type name array. Ex: type=\"sxt104\" or type=\"stx104\",\"tracer\"" );
341
342 /*****************************************************************************/
343 /*          < < < IOCTL() SUPPORT FUNCTIONS > > >                            */
344 /*****************************************************************************/
345
346 const struct file_operations auldrv_file_operations;
347
348 static int AulDrv_Initialize_Parameters_Device( struct auldrv_device_params *   dp,
349                                                 unsigned                       index_device,
350                                                 struct class *                 driver_class,
351                                                 dev_t                          driver_number,
352                                                 struct file_operations *       fops
353                                               );
354
355 static void AulDrv_Terminate_Device( struct auldrv_device_params *  dp,
356                                      struct class *                 driver_class
357                                    );
358
```

```
359 /*****************************************************************************/
360 /* TODO: ought to become IO_Definition_Get_Command()??? */
361 static int AulDrv_Ioctl_Device_Definition_Get( unsigned long argument )
362 {
363     int                     error_code;
364     int                     valid;
365     unsigned                index;
366     io_definition_type      io_definition_local;
367 
368     PDEBUG( "%s entry.\n", __func__ );
369 
370     /* get the io_definition from user space */
371     if ( IO_COPY_FROM_USER( &io_definition_local, argument, sizeof( io_definition_local ) ) )
372     { error_code = EC_IO_COPY_FROM_USER;    goto AulDrv_Ioctl_Device_Definition_Get_Error;   }
373 
374     /* search for definition by comparing acronyms */
375     valid = -1; /* assume invalid */
376     for ( index = 0; index < auldrv.device_count; index++ )
377     {
378         if ( 0 == String_Compare( auldrv.device[index].region.acronym,
379                                   io_definition_local.acronym,
380                                   IO_DEFINITION_ACRONYM_SIZE          )      )
381         {
382             valid = index;
383             break;
384         }
385     }
386     if ( -1 != valid )
387     {   /* pass it back to the user with additional information */
388         if ( IO_COPY_TO_USER( argument, &(auldrv.device[valid].region), sizeof( io_definition_type ) ) )
389         { error_code = EC_IO_COPY_TO_USER;  goto AulDrv_Ioctl_Device_Definition_Get_Error;  }
390     }
391     else
392     { error_code = EC_AULDRV_IOCTL_DEFINITION;  goto AulDrv_Ioctl_Device_Definition_Get_Error;  }
393 
394     PDEBUG( "%s exit success.\n", __func__ );
395     return SUCCESS;
396 AulDrv_Ioctl_Device_Definition_Get_Error:
397     if ( error_code < 0 ) error_code = -error_code;
398     PDEBUG( "%s exit fail, error = %d.\n", __func__, error_code );
399     return -error_code;
```

```
400 }
401 /*****************************************************************************/
402 /*  */
403 static int AulDrv_Ioctl_Device_List_Get( unsigned long argument )
404 {
405     int                     error_code;
406     unsigned                index;
407     /* NOTE: struct device_list same as device_list_type */
408     device_list_type        device_list;  /* null terminated list */
409 
410 
411     Mem_Set( &device_list, 0, sizeof( device_list ) );
412     device_list.type = SYS_TYPE_MAKE( device_list_type );
413     device_list.region_count = (size_t) auldrv.device_count;
414 
415     index = 0;
416     while( index < auldrv.device_count )
417     {
418         Mem_Copy( &(device_list.region[index]), &(auldrv.device[index].region), sizeof( device_list.region[index] ) );
419         device_list.open_and_active[index]  = auldrv.device[index].open_and_active;
420         index++;
421     }
422     if ( IO_COPY_TO_USER( argument, &(device_list), sizeof( device_list ) ) )
423     { error_code = EC_IO_COPY_TO_USER;  goto AulDrv_Ioctl_Device_List_Get_Error;  }
424     PDEBUG( "%s exit success.\n", __func__ );
425     return SUCCESS;
426 AulDrv_Ioctl_Device_List_Get_Error:
427     if ( error_code < 0 ) error_code = -error_code;
428     PDEBUG( "%s exit fail, error = %d.\n", __func__, error_code );
429     return -error_code;
430 }
431 /*****************************************************************************/
432 
433 
434 
435 /*****************************************************************************/
436 /*****************************************************************************/
437 /*****************************************************************************/
438 /*          < < < IOCTL() FUNCTION > > >                                     */
439 /*****************************************************************************/
440 #if (LINUX_VERSION_CODE < KERNEL_VERSION(2,6,35))
```

```
441 static int AulDrv_Ioctl(    struct inode * inode,
442                              struct file *   file,
443                              unsigned int    command,
444                              unsigned long   argument
445                          )
446 {
447 #else
448 static long AulDrv_Ioctl(   struct file *   file,
449                              unsigned int    command,
450                              unsigned long   argument
451                           )
452 {
453     struct inode *  inode       = file->f_dentry->d_inode;
454 #endif
455     int         error_code;
456     unsigned    index_device;
457     IOC_T       module_logical_id;
458     unsigned    minor = MINOR( inode->i_rdev );
459
460     PDEBUG( "%s entry.\n", __func__ );
461
462     if ( !IOC_MAGIC_IS_VALID( command )  )
463     {   error_code = EC_AULDRV_IOCTL_MAGIC_NUMBER;  goto AulDrv_Ioctl_Error;    }
464
465     module_logical_id = IOC_GET_LOGICAL_ID_MODULE( command );
466     error_code = SUCCESS;
467     if ( ( AULDRV_LOGICAL_ID == module_logical_id ) && ( AULDRV_DRIVER_MINOR_NUMBER == minor ) )
468     { /* controller */
469
470         switch ( command )
471         {
472             case CMD_AULDRV_VERSION:
473                 {
474                     uint32_t version;
475                     version = TSSVN_YYMMDDHH_DEC( AULDRV_MODULE_REV );
476                     if ( IO_COPY_TO_USER( argument, &version, sizeof( version ) ) )
477                     { error_code = EC_AULDRV_COPY_TO_USER;  goto AulDrv_Ioctl_Error;  }
478                 }
479                 break;
480             case CMD_AULDRV_NAME:
481                 {
```

```
482                     if ( IO_COPY_TO_USER( argument, auldrv_ioctl_module_name, String_Length( auldrv_ioctl_module_name ) + 1 )
    )
483                     { error_code = EC_AULDRV_COPY_TO_USER;  goto AulDrv_Ioctl_Error;  }
484                 }
485                 break;
486             case CMD_AULDRV_DEVICE_DEFINITION_AVAILABLE:
487                 /* TODO: ought to become IO_Definition_Available_Command()??? */
488                 error_code = SUCCESS;
489                 break;
490             case CMD_AULDRV_DEVICE_DEFINITION_GET:
491                 /* TODO: ought to become IO_Definition_Get_Command()??? */
492                 error_code = AulDrv_Ioctl_Device_Definition_Get( argument );
493                 break;
494             case CMD_AULDRV_DEVICE_LIST_GET:
495                 /* retrieve all devices information (mostly generated at insmod)  */
496                 error_code = AulDrv_Ioctl_Device_List_Get( argument );
497                 break;
498             case CMD_AULDRV_DEBUG_ACTIVE_GET:
499                 { /* return zero if _DEBUG is not defined, otherwise non-zero */
500                     int debug_active;
501 #ifdef _DEBUG
502                     debug_active = 1;
503 #else
504                     debug_active = 0;
505 #endif
506                     if ( IO_COPY_TO_USER( argument, &debug_active, sizeof( debug_active ) ) )
507                     { error_code = EC_AULDRV_COPY_TO_USER;  goto AulDrv_Ioctl_Error;  }
508                 }
509                 break;
510 #ifdef AULDRV_DEVICE_DYNAMIC
511             case 1033: /* create stx104 */
512                 {
513                     unsigned index_device = argument;
514                     /* TODO: test for argument validity */
515                     error_code = AulDrv_Initialize_Parameters_Device( &(auldrv.device[index_device]),
516                                                                       index_device,
517                                                                       auldrv.driver.class,
518                                                                       auldrv.driver.number,
519                                                                       &auldrv_file_operations
520                                                                     );
521                     if ( error_code ) goto AulDrv_Ioctl_Error;
```

```
522                     auldrv.device_count++;
523                     minor = index_device + 1;
524                     auldrv.map_minor_to_device[minor] = index_device;
525
526                     error_code = IO_Open( &(auldrv.device[index_device].io_descriptor), &(auldrv.device[index_device].region)
    );
527                     if ( error_code ) goto AulDrv_Ioctl_Error;
528                     error_code = IO_Capture( auldrv.device[index_device].io_descriptor );
529                     if ( error_code ) goto AulDrv_Ioctl_Error;
530                 }
531                 break;
532             case 1099: /* destroy stx104 */
533                 {
534                     unsigned index_device = argument;
535                     /* TODO: test for argument validity */
536                     AulDrv_Terminate_Device( &(auldrv.device[index_device]), auldrv.driver.class );
537
538                     minor = index_device + 1;
539                     auldrv.map_minor_to_device[minor] = AULDRV_DRIVER_MINOR_INVALID;
540                     auldrv.device_count--;
541
542                     error_code = IO_Release( auldrv.device[index_device].io_descriptor, 1 /* test_for_device_ready */ );
543                     if ( error_code ) goto AulDrv_Ioctl_Error;
544                     error_code = IO_Close( &(auldrv.device[index_device].io_descriptor) );
545                     if ( error_code ) goto AulDrv_Ioctl_Error;
546                 }
547                 break;
548 #endif
549             default:
550                 error_code = EC_AULDRV_IOCTL_COMMAND_UNKNOWN; /* command invalid */
551         }
552     }
553     else if ( IO_LOGICAL_ID == module_logical_id )
554     {
555         if ( !auldrv.driver.open_and_active )
556         { error_code = EC_AULDRV_DRIVER_OPEN_NOT; goto AulDrv_Ioctl_Error; }
557         /* TODO: test for argument validity */
558         index_device = auldrv.map_minor_to_device[minor];
559         /* perform the actual I/O command requested */
560         error_code   = IO_Ioctl( auldrv.device[index_device].io_descriptor, command, argument );
561         if ( error_code ) goto AulDrv_Ioctl_Error;
```

```
562     }
563     else
564     {
565         error_code = EC_AULDRV_IOCTL_COMMAND_UNKNOWN; /* command invalid */
566         goto AulDrv_Ioctl_Error;
567     }
568     if ( error_code ) goto AulDrv_Ioctl_Error;
569     PDEBUG( "%s exit success.\n", __func__ );
570     return SUCCESS;
571
572 AulDrv_Ioctl_Error:
573     if ( error_code < 0 ) error_code = -error_code;
574     PDEBUG( "%s exit fail, error = %d.\n", __func__, error_code );
575     return -error_code;
576 }
577
578 /*****************************************************************************/
579 /*****************************************************************************/
580 /*****************************************************************************/
581 /*    < < < OPEN() AND CLOSE() FUNCTION > > >                                */
582 /*****************************************************************************/
583
584 static atomic_t auldrv_driver_available = ATOMIC_INIT(1);
585
586 /*****************************************************************************//**
587  *
588  **/
589 static int AulDrv_Open( struct inode * inode, struct file * file )
590 {
591     int         error_code = SUCCESS;
592     unsigned    minor = MINOR( inode->i_rdev );
593     unsigned    index_device;
594
595     PDEBUG( "%s entry.\n", __func__ );
596
597     if ( AULDRV_DRIVER_MINOR_NUMBER == minor )
598     { /* controller */
599
600         if ( ! atomic_dec_and_test( &auldrv_driver_available ) )
601         {
602             atomic_inc( &auldrv_driver_available );
```

```
603             return -EBUSY;
604         }
605 
606 
607         PDEBUG( " %s:  minor = %d, controller = %s\n", __func__, minor, auldrv_module_name );
608         if ( auldrv.driver.open_and_active )
609         { error_code = EC_AULDRV_DRIVER_ALREADY_OPEN; goto AulDrv_Open_Error; }
610         //file->private_data = &(auldrv.device);
611         auldrv.driver.open_and_active = 1;
612     }
613     else
614     {
615         index_device = auldrv.map_minor_to_device[minor];
616         if ( AULDRV_DRIVER_MINOR_INVALID == index_device )
617         { error_code = EC_AULDRV_OPEN_MINOR_INVALID;   goto AulDrv_Open_Error;  }
618 
619         if ( auldrv.device[index_device].open_and_active )
620         { error_code = EC_AULDRV_DRIVER_ALREADY_OPEN; goto AulDrv_Open_Error; }
621 
622         PDEBUG( " %s: minor = %d, device = %s\n", __func__, minor, auldrv.device[index_device].region.acronym );
623 
624         error_code = IO_Open( &(auldrv.device[index_device].io_descriptor), &(auldrv.device[index_device].region) );
625         if ( error_code ) goto AulDrv_Open_Error;
626 
627         auldrv.device[index_device].open_and_active = 1;
628     }
629     PDEBUG( "%s exit success.\n", __func__ );
630     return SUCCESS;
631 AulDrv_Open_Error:
632     if ( error_code < 0 ) error_code = -error_code;
633     PDEBUG( "%s exit fail, error = %d.\n", __func__, error_code );
634     return -error_code;
635 }
636 /*****************************************************************************//**
637  *
638  **/
639 static int AulDrv_Close( struct inode * inode, struct file * file )
640 {
641     int         error_code;
642     unsigned    minor = MINOR( inode->i_rdev );
643     unsigned    index_device;
```

```
644
645     PDEBUG( "%s entry.\n", __func__ );
646     if ( AULDRV_DRIVER_MINOR_NUMBER == minor )
647     { /* controller */
648
649         atomic_inc( &auldrv_driver_available );
650
651         PDEBUG( " %s: minor = %d, controller = %s\n", __func__, minor, auldrv_module_name );
652         auldrv.driver.open_and_active = 0;
653     }
654     else
655     {
656         index_device = auldrv.map_minor_to_device[minor];
657         PDEBUG( " %s: minor = %d, device = %s\n", __func__, minor, auldrv.device[index_device].region.acronym );
658 #if(0)
659 /* Ioctl() -> IO_Ioctl() by library */
660         error_code = IO_Release( auldrv.device[index_device].io_descriptor, 0 /* test_for_device_ready */ );
661         if ( error_code ) goto AulDrv_Close_Error;
662 #endif
663
664         error_code = IO_Close( &(auldrv.device[index_device].io_descriptor) );
665         if ( error_code ) goto AulDrv_Close_Error;
666
667         auldrv.device[index_device].open_and_active = 0;
668     }
669     PDEBUG( "%s exit success.\n", __func__ );
670     return SUCCESS;
671 AulDrv_Close_Error:
672     if ( error_code < 0 ) error_code = -error_code;
673     PDEBUG( "%s exit fail, error = %d.\n", __func__, error_code );
674     return -error_code;
675 }
676
677 /*****************************************************************************/
678 /*****************************************************************************/
679 /*****************************************************************************/
680 /*    < < < FILE OPERATIONS DEFINITION > > >                                 */
681 /*****************************************************************************/
682 const struct file_operations auldrv_file_operations =
683 {
684     .owner                      = THIS_MODULE,      /* owner                 */
```

```
685     .open                       = AulDrv_Open,      /* open method                    */
686     .release                    = AulDrv_Close,     /* release or close method        */
687 //  .read                       = AulDrv_Read,      /* read method                    */
688 //  .write                      = AulDrv_Write,     /* write method                   */
689 //  .llseek                     = device_seek,      /* seek method                    */
690 //  .poll
691 #if (LINUX_VERSION_CODE < KERNEL_VERSION(2,6,35))
692     .ioctl                      = AulDrv_Ioctl
693 #else
694     .unlocked_ioctl             = AulDrv_Ioctl      /* Note: unlocked_ioctl avoids Big Kernel Locking (BKL) */
695 #endif
696 };
697
698 /*****************************************************************************/
699 /*****************************************************************************/
700 /*****************************************************************************/
701 /*   < < < MODULE INITIALIZE AND EXIT FUNCTIONS > > >                        */
702 /*****************************************************************************/
703
704 /*****************************************************************************/
705 static void AulDrv_Terminate_Device( struct auldrv_device_params *  dp,
706                                      struct class *                 driver_class
707                                    )
708 {
709     if ( NULL != driver_class ) device_destroy( driver_class, dp->number );
710     if ( NULL != dp->cdev.ops ) cdev_del( &(dp->cdev) );
711     /* TODO: clear all device parameters */
712 }
713 /*****************************************************************************/
714 static void AulDrv_Terminate_Devices( struct auldrv_device_params * dp,
715                                       unsigned                      count,
716                                       struct class *                driver_class
717                                     )
718 {
719     unsigned    index_device;
720
721     for ( index_device = 0; index_device < count; index_device++ )
722     {
723         AulDrv_Terminate_Device( &(dp[index_device]), driver_class );
724     }
725 }
```

```
726 /*****************************************************************************/
727 static void AulDrv_Terminate_Driver( struct auldrv_driver_params * dp )
728 {
729     /* release, if any, device class associated with driver */
730     if ( NULL != dp->class )
731     {
732         device_destroy( dp->class, dp->number );
733     }
734     /* release, if any, the cdev associated with driver */
735     if ( NULL != dp->cdev.ops )
736     {
737         cdev_del( &(dp->cdev) );
738     }
739     /* release, if any, driver class */
740     if ( NULL != dp->class )
741     {
742         class_unregister( dp->class );
743         class_destroy(    dp->class );
744     }
745     /* release, if any, the major/minor numbers */
746     if ( dp->number )
747     {
748         unregister_chrdev_region( dp->number, dp->minor_count );
749     }
750     /* TODO: clear all driver parameters */
751 }
752 /*****************************************************************************/
753 static void AulDrv_Terminate_AulDrv_Dataset( struct auldrv_dataset * ds )
754 {
755     AulDrv_Terminate_Devices( ds->device, ds->device_count, ds->driver.class );
756     AulDrv_Terminate_Driver( &(ds->driver) );
757     /* TODO: clear ds->driver */
758 }
759 /*****************************************************************************************************************//**
760 @ingroup auldrv
761 @brief
762 Return a driver number which includes both major and minor numbers.  Register the
763 device.
764
765 @return OS type error code.
766 */
```

```
767 static int AulDrv_Initialize_Device_Add( struct cdev *              cdev,
768                                           dev_t                     number,
769                                           struct file_operations *  fops
770                                         )
771 {
772     int error_code;
773     cdev_init( cdev, fops );
774     cdev->owner = THIS_MODULE;
775     cdev->ops   = fops;
776     error_code  = cdev_add( cdev, number, 1 );
777     return error_code;
778 }
779 /*****************************************************************************************************//**
780 @ingroup auldrv
781 @brief
782 Return a driver number which includes both major and minor numbers.  Register the
783 device.
784
785 @return OS type error code.
786 */
787 static int AulDrv_Initialize_Driver_Number( unsigned    major_number,
788                                             dev_t *     number,
789                                             size_t      minor_quantity,
790                                             char *      driver_name
791                                           )
792 {
793     int     error_code;
794     dev_t   driver_number;
795
796 PDEBUG( "%s entry.\n", __func__ );
797
798     if ( major_number )
799     {
800         driver_number = MKDEV( major_number, AULDRV_DRIVER_MINOR_NUMBER );
801         error_code  = register_chrdev_region( driver_number, minor_quantity, driver_name);
802         if ( !error_code ) *number = driver_number;
803     }
804     else
805     {
806         error_code  = alloc_chrdev_region( number, AULDRV_DRIVER_MINOR_NUMBER, minor_quantity, driver_name );
807     }
```

```
808     return error_code;
809 }
810 /*****************************************************************************************//**
811 @ingroup auldrv
812 @brief
813 The module definition of auldrv.  Currently, the name and revision are used for purposes of reporting
814 and used with udev to produce the device "auldrv" as seen in /dev.
815
816 @return Nothing
817 */
818 static void AulDrv_Initialize_Definition_Module( struct module_definition * d )
819 {
820 PDEBUG( "%s entry.\n", __func__ );
821
822     d->initialize       = NULL;
823     d->terminate        = NULL;
824     d->name             = auldrv_module_name;
825     d->svn_module_date  = AULDRV_MODULE_REV;
826     d->svn_command_date = NULL;
827     d->svn_status_date  = NULL;
828 }
829 /*****************************************************************************************//**
830 @ingroup auldrv
831 @brief
832 This is an internal initialization function.  Populates the io_definition based on the
833 insmod command line parameters entered.  If the requested type of device is not found
834 then
835
836 @return
837 An error code of type int is returned. If successful, returns 0 (SUCCESS), otherwise it
838 returns a non-zero error code. Please refer to ::AUL_STATUS_ENUM for all the possible error codes.
839 */
840 static int AulDrv_Initialize_Driver( struct auldrv_driver_params *  dp,
841                                      struct class *                 driver_class,
842                                      dev_t                          driver_number,
843                                      struct file_operations *       fops
844                                    )
845 {
846     int             error_code;
847     struct device * dev_ret;
848 PDEBUG( "%s entry.\n", __func__ );
```

```
849     dp->major           = MAJOR( driver_number );
850     dp->minor           = MINOR( driver_number );
851     dp->number          = driver_number;
852     dp->name            = dp->module_aul.name;
853     dp->minor_count     = AULDRV_REGION_QTY + 1;
854     dp->open_and_active = 0;
855     dp->error_code_last = SUCCESS;
856     dp->class           = driver_class;
857
858 PDEBUG( "%s:  major = %d.\n", __func__, dp->major );
859 PDEBUG( "%s:  minor = %d.\n", __func__, dp->minor );
860
861     error_code = AulDrv_Initialize_Device_Add( &(dp->cdev), dp->number, fops );
862     if ( error_code ) return error_code;
863
864 PDEBUG( "%s AulDrv_Initialize_Device_Add complete.\n", __func__ );
865
866     /* send uevents to udev, so it will create /dev node */
867     dev_ret = device_create( dp->class,     /* class  */
868                              NULL,          /* parent */
869                              dp->number,
870                              NULL,
871                              "%s",
872                              dp->name
873                            );
874     if ( NULL == dev_ret )
875     {
876         error_code = ENODEV;
877     }
878 PDEBUG( "%s device_create, error=%d.\n", __func__, error_code );
879     return error_code;
880 }
881 /*****************************************************************************/
882 /* TODO: duplicate in aul_hwdef.c - merge to a central location */
883 #if defined( DEVICE_TRACERE )
884  static const char auldrv_tracere[] = DEVICE_TRACERE_NAME;
885 #endif
886 #if defined( DEVICE_STX104ND )
887  static const char auldrv_stx104nd[] = DEVICE_STX104ND_NAME;
888 #endif
889 #if defined( DEVICE_STX104 )
```

```
890  static const char auldrv_stx104[]  = DEVICE_STX104_NAME;
891 #endif
892 #if defined( DEVICE_SUMMIT )
893  static const char auldrv_summit[]  = DEVICE_SUMMIT_NAME;
894 #endif
895 /* <EGS_BEGIN>*/
896 #if defined( DEVICE_STX104EGS )
897  static const char auldrv_stx104egs[]  = DEVICE_STX104_NAME_EGS;
898 #endif
899 /* <EGS_END>*/
900  /*****************************************************************************************//**
901  @ingroup auldrv
902  @brief
903  This is an internal initialization function.  Populates the io_definition based on the
904  insmod command line parameters entered.  If the requested type of device is not found
905  then
906
907  @return
908  An error code of type int is returned. If successful, returns 0 (SUCCESS), otherwise it
909  returns a non-zero error code. Please refer to ::AUL_STATUS_ENUM for all the possible error codes.
910  */
911 static void AulDrv_Initialize_Definition_Device( struct io_definition * d, unsigned index_device )
912 {
913 PDEBUG( "%s entry using index_device = %d\n", __func__, index_device );
914     d->region_offset        = auldrv_requested_region_offset[index_device];
915     d->region_bytes         = auldrv_requested_region_bytes[index_device];
916     d->region_restrict_8bit = auldrv_requested_region_restrict_8bit[index_device];
917     d->interrupt_number     = 0; /*TODO: */
918     d->minor_number         = index_device;
919     if ( NULL != auldrv_requested_region_acronym[index_device] )
920     {
921         strncpy( d->acronym, auldrv_requested_region_acronym[index_device], IO_DEFINITION_ACRONYM_SIZE );
922     }
923     else
924     {
925         d->acronym[0] = '\0';
926     }
927 PDEBUG( "  d->acronym = %s, ", d->acronym );
928     /* assume none */
929     d->physical_id = IOC_LOGICAL_ID_NULL;
930 #if defined( DEVICE_TRACERE )
```

```
931     if ( 0 == String_Compare( auldrv_tracere, auldrv_requested_region_type[index_device],
932                               String_Length( auldrv_tracere )                          ) )
933     {
934         d->physical_id = IOC_CREATE_LOGICAL_ID_DEVICE( DEVICE_TRACERE );
935 PDEBUG( "TRACERE: d->physical_id = %d\n", d->physical_id );
936     }
937 #endif
938 #if defined( DEVICE_STX104ND )
939     if( 0 == String_Compare( auldrv_stx104nd, auldrv_requested_region_type[index_device],
940                              String_Length( auldrv_stx104nd )                               ) )
941     {
942         d->physical_id = IOC_CREATE_LOGICAL_ID_DEVICE( DEVICE_STX104ND );
943 PDEBUG( "STX104ND: d->physical_id = %d\n", d->physical_id );
944     }
945 #endif
946 #if defined( DEVICE_STX104 )
947     if( 0 == String_Compare( auldrv_stx104, auldrv_requested_region_type[index_device],
948                              String_Length( auldrv_stx104 )                             ) )
949     {
950         d->physical_id = IOC_CREATE_LOGICAL_ID_DEVICE( DEVICE_STX104 );
951 PDEBUG( "STX104: d->physical_id = %d\n", d->physical_id );
952     }
953 #endif
954 #if defined( DEVICE_SUMMIT )
955     if ( 0 == String_Compare( auldrv_summit, auldrv_requested_region_type[index_device],
956                               String_Length( auldrv_summit )                           ) )
957     {
958         d->physical_id = IOC_CREATE_LOGICAL_ID_DEVICE( DEVICE_SUMMIT );
959 PDEBUG( "SUMMIT: d->physical_id = %d\n", d->physical_id );
960     }
961 #endif
962 /* <EGS_BEGIN>*/
963 #if defined( DEVICE_STX104_EGS )
964 if( 0 == String_Compare( auldrv_stx104egs, auldrv_requested_region_type[index_device],
965                          String_Length( auldrv_stx104 )                               ) )
966 {
967     d->physical_id = IOC_CREATE_LOGICAL_ID_DEVICE( DEVICE_STX104 );
968
969     // adjust byte width here.
970 PDEBUG( "STX104EGS: d->physical_id = %d\n", d->physical_id );
971 }
```

```
#endif
/* <EGS_END>*/
PDEBUG( "%s exit success.\n", __func__ );
}
/*****************************************************************************************//**
@ingroup auldrv
@brief
The main initialization function used during installation of the driver.

@return
An error code of type int is returned. If successful, returns 0 (SUCCESS), otherwise it
returns a non-zero error code. Please refer to ::AUL_STATUS_ENUM for all the possible error codes.
*/
static int AulDrv_Initialize_Parameters_Device( struct auldrv_device_params *   dp,
                                                unsigned                        index_device,
                                                struct class *                  driver_class,
                                                dev_t                           driver_number,
                                                struct file_operations *        fops
                                              )
{
    int             error_code;
    struct device * dev_ret;
PDEBUG( "%s entry.\n", __func__ );
    AulDrv_Initialize_Definition_Device( &(dp->region), index_device );
    dp->major           = MAJOR( driver_number );
    dp->minor           = MINOR( driver_number ) + index_device + 1;
    dp->number          = MKDEV( dp->major, dp->minor );
    dp->name            = dp->region.acronym;
    dp->open_and_active = 0;

PDEBUG( "%s: index = %d, major = %d.\n", __func__, index_device, dp->major );
PDEBUG( "%s: index = %d, minor = %d.\n", __func__, index_device, dp->minor );

    error_code = AulDrv_Initialize_Device_Add( &(dp->cdev), dp->number, fops );
    if ( error_code ) return error_code;

PDEBUG( "%s AulDrv_Initialize_Device_Add complete.\n", __func__ );

    /* send uevents to udev, so it will create /dev node */
    dev_ret = device_create( driver_class,              /* class  */
                             NULL,                      /* parent */
```

```
1013                                dp->number,
1014                                NULL,
1015                                "%s",
1016                                dp->name
1017                              );
1018    if ( NULL == dev_ret )
1019    {
1020        error_code = ENODEV;
1021    }
1022    return error_code;
1023 }
1024 /*****************************************************************************************************//**
1025 @ingroup auldrv
1026 @brief
1027 The main initialization function used during installation of the driver.
1028
1029 @return
1030 An error code of type int is returned. If successful, returns 0 (SUCCESS), otherwise it
1031 returns a non-zero error code. Please refer to ::AUL_STATUS_ENUM for all the possible error codes.
1032 */
1033 static int AulDrv_Initialize_Devices( struct auldrv_device_params * dp,
1034                                       unsigned *                    device_count_inout,
1035                                       struct class *                driver_class,
1036                                       dev_t                         driver_number,
1037                                       struct file_operations *      fops,
1038                                       unsigned *                    map_minor_to_device
1039                                     )
1040 {
1041     int         error_code;
1042     unsigned    index_device;
1043     unsigned    minor;
1044     unsigned    count;
1045     unsigned    quantity;
1046 PDEBUG( "%s entry.\n", __func__ );
1047     quantity = *device_count_inout;
1048     count = 0;
1049
1050 PDEBUG( "%s: quantity = %d.\n", __func__, quantity );
1051
1052     for ( index_device = 0; index_device < quantity; index_device++ )
1053     {
```

```
1054        error_code = AulDrv_Initialize_Parameters_Device( &(dp[index_device]),
1055                                                          index_device,
1056                                                          driver_class,
1057                                                          driver_number,
1058                                                          fops
1059                                                        );
1060        if ( error_code ) return error_code;
1061        minor = dp[index_device].minor;
1062        map_minor_to_device[minor] = index_device;
1063        count++;
1064        *device_count_inout = count;
1065    }
1066    return SUCCESS;
1067 }
1068 /*****************************************************************************************************//**
1069 @ingroup auldrv
1070 @brief
1071 The main initialization function used during installation of the driver.
1072
1073 @return
1074 An error code of type int is returned. If successful, returns 0 (SUCCESS), otherwise it
1075 returns a non-zero error code. Please refer to ::AUL_STATUS_ENUM for all the possible error codes.
1076 */
1077 static int AulDrv_Initialize_Dataset( struct auldrv_dataset *   ds,
1078                                       struct class *            driver_class,
1079                                       dev_t                     driver_number,
1080                                       struct file_operations *  fops
1081                                     )
1082 {
1083    int error_code;
1084 PDEBUG( "%s entry.\n", __func__ );
1085    AulDrv_Initialize_Definition_Module( &(ds->driver.module_aul) );
1086
1087    error_code = AulDrv_Initialize_Driver(  &(ds->driver),
1088                                            driver_class,
1089                                            driver_number,
1090                                            fops
1091                                          );
1092    if ( error_code ) return error_code;
1093
1094 #ifdef AULDRV_DEVICE_DYNAMIC
```

```
1095     ds->device_count = 0;
1096 PDEBUG( "%s: device_count = %d.\n", __func__, ds->device_count );
1097 #else
1098 /* NOTE: device_count will never be larger than AULDRV_REGION_QTY, see
1099  *       AulDrv_Initialize_Driver() above.
1100  */
1101     ds->device_count = ds->driver.minor_count - 1;
1102
1103 PDEBUG( "%s: device_count = %d.\n", __func__, ds->device_count );
1104
1105     error_code = AulDrv_Initialize_Devices( ds->device,
1106                                             &(ds->device_count),
1107                                             driver_class,
1108                                             driver_number,
1109                                             fops,
1110                                             ds->map_minor_to_device
1111                                           );
1112 #endif
1113     return error_code;
1114 }
1115
1116 /*****************************************************************************
1117 TODO:  __init required?
1118
1119 group is "staff"
1120 */
1121 /*****************************************************************************************************************//**
1122 @ingroup auldrv
1123 @brief
1124 The main initialization function used during installation of the driver.
1125
1126 @return
1127 An error code of type int is returned. If successful, returns 0 (SUCCESS), otherwise it
1128 returns a non-zero error code. Please refer to ::AUL_STATUS_ENUM for all the possible error codes.
1129 */
1130 /*****************************************************************************//**
1131  * @brief
1132  * The main initialization function used during installation of the driver.
1133  **/
1134 int AulDrv_Initialize( void )
1135 {
```

```
1136    int             error_code;
1137    /* Total number of minor numbers we need for this driver.
1138       We will need a total of AULDRV_REGION_QTY + 1 where the
1139       additional one is for the driver and the AULDRV_REGION_QTY
1140       is for all the regions/devices
1141    */
1142    unsigned        minor_number_count = AULDRV_REGION_QTY + 1;
1143
1144    dev_t           driver_number;
1145    struct class *  driver_class;
1146
1147    unsigned        minor;
1148    unsigned        index_device;
1149
1150    struct device * dev_ret;
1151
1152    /* Since we use the name of the driver a lot within the
1153       initialization, it seems like a good idea to have a
1154       a pointer to it that has a name indicating what it is.
1155    */
1156    //const char *    driver_name = auldrv_module_name;
1157
1158
1159    PDEBUG( "%s entry.\n", __func__ );
1160
1161 /* PASSED IN PARAMETERS VIA INSMOD */
1162 //  AulDrv_Init_Help();
1163
1164    /* Acquire driver major number and allocate minor numbers.
1165     *  The auldrv_requested_driver_major_number is typically set to zero indicating
1166     *  that we are requesting a major number be assigned.
1167     */
1168    error_code = AulDrv_Initialize_Driver_Number(   auldrv_requested_driver_major_number,  /* typically 0 */
1169                                                    &driver_number,      /* out-bound driver number */
1170                                                    minor_number_count,  /* number of boards + 1    */
1171                                                    auldrv_module_name   /* the name of this driver */
1172                                                );
1173    if ( error_code ) goto AulDrv_Initialize_Error;
1174
1175 /*
1176 PDEBUG( "%s: CMD_AULDRV_DEVICE_DEFINITION_AVAILABLE   = %d.\n", __func__, CMD_AULDRV_DEVICE_DEFINITION_AVAILABLE );
```

```
1177 PDEBUG( "%s: CMD_AULDRV_DEVICE_DEFINITION_GET = %d.\n", __func__, CMD_AULDRV_DEVICE_DEFINITION_GET );
1178 PDEBUG( "%s: CMD_AULDRV_VERSION               = %d.\n", __func__, CMD_AULDRV_VERSION );
1179 */
1180
1181 PDEBUG( "%s: AulDrv_Initialize_Driver_Number complete.\n", __func__ );
1182 PDEBUG( "%s:  major = %d.\n", __func__, MAJOR(driver_number) );
1183 PDEBUG( "%s:  minor = %d.\n", __func__, MINOR(driver_number) );
1184
1185     /* create the driver class */
1186     driver_class = class_create( THIS_MODULE, auldrv_module_name );
1187     if ( NULL == driver_class )
1188     {   error_code = ENODEV;        goto AulDrv_Initialize_Error;   }
1189
1190 PDEBUG( "%s: class_create complete.\n", __func__ );
1191
1192     /* Initialize the driver/controller and the devices/regions */
1193     error_code = AulDrv_Initialize_Dataset( &auldrv, driver_class, driver_number, &auldrv_file_operations );
1194     if ( error_code ) goto AulDrv_Initialize_Error;
1195
1196 PDEBUG( "%s: AulDrv_Initialize_Dataset complete.\n", __func__ );
1197
1198     /* Initialize I/O behavior.  For Linux this will be either io_linux_drv_gen_x86 or
1199      *  io_linux_lib_sim, depending on what is compiled based on the make file.  This
1200      *  allows us great flexibility in terms of how data will flow to/from the hardware.
1201      */
1202     error_code = IO_Initialize();
1203     if ( error_code ) goto AulDrv_Initialize_Error;
1204
1205     PDEBUG( "%s exit success.\n", __func__ );
1206
1207     return SUCCESS;
1208 AulDrv_Initialize_Error:
1209     if ( error_code < 0 ) error_code = -error_code;
1210     PDEBUG( "%s fail, error = %d.\n", __func__, error_code );
1211     AulDrv_Terminate_AulDrv_Dataset( &auldrv );
1212     return -error_code;
1213 }
1214
1215 /*****************************************************************
1216 * The cleanup_module()
1217 */
```

```
1218 /*****************************************************************************//**
1219  * @brief
1220  *
1221  **/
1222 void AulDrv_Terminate( void )
1223 {
1224     PDEBUG( "%s entry.\n", __func__ );
1225     AulDrv_Terminate_AulDrv_Dataset( &auldrv );
1226     IO_Terminate();
1227     PDEBUG( "%s exit success.\n", __func__ );
1228 }
1229
1230 /*****************************************************************************/
1231 /*****************************************************************************/
1232 /*****************************************************************************/
1233 /*    < < < MODULE DEFINITIONS > > >                                         */
1234 /*****************************************************************************/
1235
1236 module_init(AulDrv_Initialize);
1237 module_exit(AulDrv_Terminate);
1238
1239 MODULE_AUTHOR( "customer.service@apexembeddedsystems.com" );
1240 MODULE_DESCRIPTION("AES Generic Kernel Driver");
1241 //TODO: MODULE_VERSION();
1242 MODULE_LICENSE("GPL");  /* for now */
1243
```