```c
/*
 * idi.c
 *
 *  Created on: Feb 25, 2015
 *      Author: Mike
 */


/* TODO:
 *
 * 2. comments above function definitions and any other definitions not complete.
 *
 *
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>


#if defined( __MSDOS__ )
# include <mem.h>
# include <dos.h>    /* outportb() and inportb() */
# include <conio.h>  /* kbhit() and getch()      */
#else
# ifndef strcmpi
#  define strcmpi strcasecmp
# endif
# include <stdint.h>
#endif


/**************************************************************************************************//**
 * @ingroup idi
 * @brief
 * The Subversion (SVN) time/date marker which is updated during commit of the
 * source file to the repository.
 */
#define IDI_REV  "$Date: 2015-03-05 22:16:06 -0600 (Thu, 05 Mar 2015) $"


/**************************************************************************************************//**
 * @ingroup idi
 * @brief
 * The C89 compiler is typically void of these definitions, so we include them here.
 * Defines specific data width information.  The idea is to make this target independent.
 */
#if defined( __MSDOS__ )
 typedef unsigned char uint8_t;
 typedef unsigned int  uint16_t;  /* DOS only */
 typedef unsigned long uint32_t;  /* DOS only */

 typedef char int8_t;
 typedef int  int16_t;  /* DOS only */
 typedef long int32_t;  /* DOS only */
#endif
/**************************************************************************************************//**
 * @ingroup idi
 * @brief
 * Boolean logic definitions
 */
typedef int BOOL;
#define false 0
#define true  1
enum { FALSE = 0, TRUE = 1 };

/**************************************************************************************************//**
 * @ingroup idi
 * @brief
 * Board clock period as defined by the on-board oscillator which is 50MHz
 */
#define CLOCK_PERIOD_SEC   20.0e-9

/**************************************************************************************************//**
 * @ingroup idi
 * @brief
 * These are the unique IDs assigned to the hardware components.  If there is a firmware revision within
 * any of these components within the board, a new ID will be assigned.  The philosophy behind the ID scheme
 * is that it embodies both a unique ID and revision information in a purely arbitrary scheme.  It assumes
 * that all components defined within the system will never have the same ID numbers.  We maintain a unique
 * list of those ID numbers.  We will provide a list to customers as required.
 *
 * Software uses these ID numbers to determine which portion of the driver/library to apply to that portion
 * of the hardware. For example, if we end up having DINs with different ID numbers, this simply means that
 * the each will require (potentially) a unique portion of driver/library in order to properly use that
 * portion of the hardware.  This requires that the library/driver has some knowledge related to the ID numbers
 * assigned.
 *
 * At this point, an ID=0x8012 implies an Isolated Digital Input component that has a register map identical
 * to the WinSystems Opto48 card.  The actual DIN48 component has a more straight forward register mapping, but within the
 * STD-bus space we have re-arranged it to appear like the WinSystems Opto48 card.
 *
 * Similarly with the SPI component, we have mangled the register set in order that it fit within the constrained
```

```
 94  * register space.
 95  *
 96  * One could argue that there ought to be a nibble for revision and an upper nibble for the component ID itself,
 97  * but quickly you find out the limitation of that implementation.  If we simply use an arbitrary list from 1 to 65535
 98  * we can fit it within 16-bits and it will be a fairly long time until it is filled.  In a couple years we will
 99  * likely move to 32-bit and simply continue where we left off.  The software really does not care, so long as the
100  * numbers are always unique.  We can use the uniqueness of the IDs along with some implied intelligence
101  * (i.e. what board is this, what bus are we on, and perhaps a little bit of knowledge of the component itself)
102  * we can always build and grow software to accommodate new revisions and still support older revision hardware.
103  *
104  */
105 enum
106 {
107     ID_DIN = 0x8012,  /**< component DIN48 ID   */
108     ID_SPI = 0x8013   /**< component SPI   ID   */
109 };
110
111 /***
112  * Work around to test hardware that may or may not be present...useful for
113  * software debugging without any target hardware.
114  */
115 #define ID_ALWAYS_REPORT_AS_GOOD  1
116
117 /*****************************************************************************************************/
118 /*****************************************************************************************************/
119 /*****************************************************************************************************/
120
121
122
123 /*****************************************************************************************************//**
124  * @ingroup idi
125  * @brief
126  * An organized error code listing.  This macro is used to build the complete error code
127  * enumeration list.
128  *
129  */
130 #define IDI_ERROR_CODES(_) \
131     /*    enum_symbol                    code     human readable            */ \
132         _( SUCCESS,                       0,      ""                         ) \
133         _( EC_BUFFER_TOO_LARGE,           1,      "buffer too large"         ) \
134         _( EC_DIRECTION,                  2,      "direction"                ) \
135         _( EC_PARAMETER,                  3,      "parameter"                ) \
136         _( EC_NOT_FOUND,                  4,      "not found"                ) \
137         _( EC_PARAMETER_MISSING,          5,      "missing parameter"        ) \
138         _( EC_SYNTAX,                     6,      "syntax error"             ) \
139         _( EC_HEX_DUMP_COUNT,             7,      "hex dump count"           ) \
140         _( EC_INIT_FILE,                  8,      "write init file failed"   ) \
141         /* EC_BUSY emulates a similar Linux error */\
142         _( EC_BUSY,                      16,      "interrupt busy"           ) \
143         _( EC_INTERRUPT_UNAVAILABLE,     21,      "interrupt not available"  ) \
144         /* was EC_INTR_ERROR was EC_EINVAL to emulate Linux behavior */\
145         _( EC_INTR_ERROR,                22,      "interrupt error"          ) \
146         _( EC_SPI_ECD_OUT_OF_RANGE,      40,      "SPI ECD range"            ) \
147         _( EC_SPI_HALF_CLOCK_OUT_OF_RANGE, 41,    "SPI half clock range"     ) \
148         _( EC_SPI_CSB_OUT_OF_RANGE,      42,      "SPI CSB range"            ) \
149         _( EC_SPI_NOT_FOUND,             43,      "SPI not found"            ) \
150         _( EC_SPI_BUFFER_SIZE_ODD,       44,      "buffer size odd"          ) \
151         _( EC_SPI_BUFFER_SIZE,           45,      "buffer size out of range" ) \
152         _( EC_SPI_OBJECT_SIZE,           46,      "spi tx/rx object size"    )
153
154
155 /*****************************************************************/
156 #define EC_EXTRACT_ENUM(symbol,code,message)    symbol = code,
157 #define EC_EXTRACT_HUMAN_READABLE(symbol,code,message)  { code, message },
158 #define EC_HUMAN_READABLE_TERMINATE { 0, NULL }
159
160 //#define EC_EXTRACT_DEFINITION(symbol,code,message) { read_write, bank, offset, name },
161 //#define EC_DEFINITION_NONE { SUCCESS, 0, "" }
162
163 typedef enum
164 {
165     IDI_ERROR_CODES( EC_EXTRACT_ENUM )
166 } EC_ENUM;  /* EC = Error Code */
167
168 struct ec_human_readable
169 {
170     EC_ENUM         error_code;
171     const char *    message;
172 };
173
174
175
176 /*****************************************************************************************************/
177 /*****************************************************************************************************/
178 /*****************************************************************************************************/
179
180 /*****************************************************************************************************//**
181  * @ingroup idi
182  * @brief
183  * The bank register mapping.  This mapping is upwardly compatible with the legacy hardware
184  * banking register.  It also allows for future expansion utilizing the lower bits which are
185  * currently unused.
186  */
```

```
187 typedef enum
188 {
189     IDI_BANK_0          = 0x00,
190     IDI_BANK_1          = 0x40,
191     IDI_BANK_2          = 0x80,
192     IDI_BANK_3          = 0xC0,
193     IDI_BANK_4          = 0x20,
194     IDI_BANK_5          = 0x60,
195     IDI_BANK_6          = 0xA0,
196     IDI_BANK_7          = 0xE0,
197     IDI_BANK_NONE       = 0xFE, /**< indicates exclusive of bank address        */
198     IDI_BANK_UNDEFINED  = 0xFF  /**< indicates that no banking has been defined */
199 } IDI_BANK_ENUM;
200
201 typedef enum
202 {
203     REG_DIR_NONE        = 0x00,
204     REG_DIR_READ        = 0x01,
205     REG_DIR_WRITE       = 0x02,
206     REG_DIR_READ_WRITE  = 0x03
207 } REG_DIR_ENUM;
208
209
210
211 /*****************************************************************************************************//**
212  * @ingroup idi
213  * @brief
214  * Organized list of registers and associate attributes of each of the registers.  This macro does not
215  * consume any memory of in itself, and is only 'consumed' or used to automatically build enumerations
216  * and pre-built data structures.
217  */
218 #define IDI_REGISTER_SET_DEFINITION(_) \
219 /*    enum_symbol           logical   physical  byte   bus byte                                    */ \
220 /*    enum_symbol           address   offset    width  aperture   read/write      acronym        bank         */ \
221     /** DATA REGISTERS ***/ \
222     _( IDI_DI_GROUP0,           0,     0,      1,      1,      REG_DIR_READ,       "dig0",     IDI_BANK_NONE   ) \
223     _( IDI_DI_GROUP1,           1,     1,      1,      1,      REG_DIR_READ,       "dig1",     IDI_BANK_NONE   ) \
224     _( IDI_DI_GROUP2,           2,     2,      1,      1,      REG_DIR_READ,       "dig2",     IDI_BANK_NONE   ) \
225     _( IDI_DI_GROUP3,           3,     3,      1,      1,      REG_DIR_READ,       "dig3",     IDI_BANK_NONE   ) \
226     _( IDI_DI_GROUP4,           4,     4,      1,      1,      REG_DIR_READ,       "dig4",     IDI_BANK_NONE   ) \
227     _( IDI_DI_GROUP5,           5,     5,      1,      1,      REG_DIR_READ,       "dig5",     IDI_BANK_NONE   ) \
228     /** STATUS REGISTER ***/ \
229     _( IDI_INTR_BY_GROUP,       6,     6,      1,      1,      REG_DIR_READ,       "isbg",     IDI_BANK_NONE   ) \
230     /** DATA / CONTROL REGISTERS ***/ \
231     _( IDI_PEND_GROUP0,         7,     8,      1,      1,      REG_DIR_READ,       "p0",       IDI_BANK_3      ) \
232     _( IDI_PEND_GROUP1,         8,     9,      1,      1,      REG_DIR_READ,       "p1",       IDI_BANK_3      ) \
233     _( IDI_PEND_GROUP2,         9,    10,      1,      1,      REG_DIR_READ,       "p2",       IDI_BANK_3      ) \
234     _( IDI_PEND_GROUP3,        10,    11,      1,      1,      REG_DIR_READ,       "p3",       IDI_BANK_3      ) \
235     _( IDI_PEND_GROUP4,        11,    12,      1,      1,      REG_DIR_READ,       "p4",       IDI_BANK_3      ) \
236     _( IDI_PEND_GROUP5,        12,    13,      1,      1,      REG_DIR_READ,       "p5",       IDI_BANK_3      ) \
237     _( IDI_CLEAR_GROUP0,       13,     8,      1,      1,      REG_DIR_WRITE,      "c0",       IDI_BANK_3      ) \
238     _( IDI_CLEAR_GROUP1,       14,     9,      1,      1,      REG_DIR_WRITE,      "c1",       IDI_BANK_3      ) \
239     _( IDI_CLEAR_GROUP2,       15,    10,      1,      1,      REG_DIR_WRITE,      "c2",       IDI_BANK_3      ) \
240     _( IDI_CLEAR_GROUP3,       16,    11,      1,      1,      REG_DIR_WRITE,      "c3",       IDI_BANK_3      ) \
241     _( IDI_CLEAR_GROUP4,       17,    12,      1,      1,      REG_DIR_WRITE,      "c4",       IDI_BANK_3      ) \
242     _( IDI_CLEAR_GROUP5,       18,    13,      1,      1,      REG_DIR_WRITE,      "c5",       IDI_BANK_3      ) \
243     /** CONTROL REGISTERS ***/ \
244     _( IDI_BANK,               19,     7,      1,      1,      REG_DIR_READ_WRITE, "bank",     IDI_BANK_NONE   ) \
245     _( IDI_ID_LSB,             20,    14,      1,      1,      REG_DIR_READ,       "idlsb",    IDI_BANK_NONE   ) \
246     _( IDI_ID_MSB,             21,    15,      1,      1,      REG_DIR_READ,       "idmsb",    IDI_BANK_NONE   ) \
247     _( IDI_ZERO0,              22,     8,      1,      1,      REG_DIR_READ,       "zb0",      IDI_BANK_0      ) \
248     _( IDI_ZERO1,              23,     9,      1,      1,      REG_DIR_READ,       "zb1",      IDI_BANK_0      ) \
249     _( IDI_ZERO2,              24,    10,      1,      1,      REG_DIR_READ,       "zb2",      IDI_BANK_0      ) \
250     _( IDI_ZERO3,              25,    11,      1,      1,      REG_DIR_READ,       "zb3",      IDI_BANK_0      ) \
251     _( IDI_ZERO4,              26,    12,      1,      1,      REG_DIR_READ,       "zb4",      IDI_BANK_0      ) \
252     _( IDI_ZERO5,              27,    13,      1,      1,      REG_DIR_READ,       "zb5",      IDI_BANK_0      ) \
253     /** CONFIG REGISTERS ***/ \
254     _( IDI_EDGE_GROUP0,        28,     8,      1,      1,      REG_DIR_READ_WRITE, "ep0",      IDI_BANK_1      ) \
255     _( IDI_EDGE_GROUP1,        29,     9,      1,      1,      REG_DIR_READ_WRITE, "ep1",      IDI_BANK_1      ) \
256     _( IDI_EDGE_GROUP2,        30,    10,      1,      1,      REG_DIR_READ_WRITE, "ep2",      IDI_BANK_1      ) \
257     _( IDI_EDGE_GROUP3,        31,    11,      1,      1,      REG_DIR_READ_WRITE, "ep3",      IDI_BANK_1      ) \
258     _( IDI_EDGE_GROUP4,        32,    12,      1,      1,      REG_DIR_READ_WRITE, "ep4",      IDI_BANK_1      ) \
259     _( IDI_EDGE_GROUP5,        33,    13,      1,      1,      REG_DIR_READ_WRITE, "ep5",      IDI_BANK_1      ) \
260     _( IDI_INTR_BIT_GROUP0,    34,     8,      1,      1,      REG_DIR_READ_WRITE, "ibe0",     IDI_BANK_2      ) \
261     _( IDI_INTR_BIT_GROUP1,    35,     9,      1,      1,      REG_DIR_READ_WRITE, "ibe1",     IDI_BANK_2      ) \
262     _( IDI_INTR_BIT_GROUP2,    36,    10,      1,      1,      REG_DIR_READ_WRITE, "ibe2",     IDI_BANK_2      ) \
263     _( IDI_INTR_BIT_GROUP3,    37,    11,      1,      1,      REG_DIR_READ_WRITE, "ibe3",     IDI_BANK_2      ) \
264     _( IDI_INTR_BIT_GROUP4,    38,    12,      1,      1,      REG_DIR_READ_WRITE, "ibe4",     IDI_BANK_2      ) \
265     _( IDI_INTR_BIT_GROUP5,    39,    13,      1,      1,      REG_DIR_READ_WRITE, "ibe5",     IDI_BANK_2      ) \
266     /** SPI REGISTERS ***/\
267     _( SPI_ID_LSB,             40,     8,      1,      1,      REG_DIR_READ,       "sidlsb",   IDI_BANK_6      ) \
268     _( SPI_ID_MSB,             41,     9,      1,      1,      REG_DIR_READ,       "sidmsb",   IDI_BANK_6      ) \
269     _( SPI_CONFIG,             42,    10,      1,      1,      REG_DIR_READ_WRITE, "scfg",     IDI_BANK_6      ) \
270     _( SPI_ECD,                43,    11,      1,      1,      REG_DIR_READ_WRITE, "secd",     IDI_BANK_6      ) \
271     _( SPI_HCI_LSB,            44,    12,      1,      1,      REG_DIR_READ_WRITE, "shclsb",   IDI_BANK_6      ) \
272     _( SPI_HCI_MSB,            45,    13,      1,      1,      REG_DIR_READ_WRITE, "shcmsb",   IDI_BANK_6      ) \
273     _( SPI_DATA,               46,     8,      1,      1,      REG_DIR_READ_WRITE, "sdata",    IDI_BANK_7      ) \
274     _( SPI_TX_STATUS,          47,     9,      1,      1,      REG_DIR_READ,       "stxf",     IDI_BANK_7      ) \
275     _( SPI_RX_STATUS,          48,    10,      1,      1,      REG_DIR_READ,       "srxf",     IDI_BANK_7      ) \
276     _( SPI_COMMIT,             49,    11,      1,      1,      REG_DIR_READ_WRITE, "scmt",     IDI_BANK_7      ) \
277     _( IDI_UNDEFINED,          50,     0,      0,      0,      REG_DIR_NONE,       "",         IDI_BANK_UNDEFINED  )
278
279
```

```
280 /*****************************************************************/
281 #define REG_LOCATION_LOGICAL_GET(location)   ( ( location >> 8   ) & 0xFF )
282 #define REG_LOCATION_CHANNEL_GET(location)   (   location          & 0xFF )
283
284 #define REG_LOCATION_SET(index,channel)      ( ( (index & 0xFF) << 8 )  |  (channel & 0xFF) )
285
286 #define REG_EXTRACT_ENUM(symbol,index,offset,register_bytes,aperture_bytes,read_write,name,bank) symbol =
    REG_LOCATION_SET(index,0),
287
288 #define REG_EXTRACT_DEFINITION(symbol,index,offset,register_bytes,aperture_bytes,read_write,name,bank) {
    REG_LOCATION_SET(index,0), read_write, bank, offset, #symbol, name },
289
290 /*****************************************************************************************************//**
291  * @ingroup idi
292  * @brief
293  * This is referred to in the code as the 'location' symbol.  It is
294  * broken down as follows:
295  *
296  * bits  width   description
297  *  3:0    4      offset
298  *  6:4    2      direction
299  * 12:7    6      logical address (i.e. the row)
300  * 15:13   3      bank
301  *
302  *
303  *
304  */
305 /*****************************************************************/
306 typedef enum
307 {
308     IDI_REGISTER_SET_DEFINITION( REG_EXTRACT_ENUM )
309 } IDI_REG_ENUM;
310
311 /*****************************************************************/
312 struct reg_definition
313 {
314     IDI_REG_ENUM     symbol;
315     REG_DIR_ENUM     direction;
316     IDI_BANK_ENUM    bank;
317     uint16_t         physical_offset;
318     char *           symbol_name;
319     char *           acronym;
320 };
321
322
323
324 /*****************************************************************/
325 typedef enum
326 {
327     IDI_CSB_SOFTWARE     = 0,
328     IDI_CSB_BUFFER       = 1,
329     IDI_CSB_uint8_t      = 2,
330     IDI_CSB_uint16_t         = 3
331 } SPI_CSB_ENUM;
332
333 /*****************************************************************/
334 enum { SPI_FIFO_SIZE = 16 };
335
336 /*****************************************************************/
337 struct spi_cfg
338 {
339     BOOL                 sdio_wrap;             /**< SDIO_WRAP      */
340     BOOL                 sdo_polarity;          /**< SDO_POL        */
341     BOOL                 sdi_polarity;          /**< SDI_POL        */
342     BOOL                 sclk_phase;            /**< SCLK_PHA       */
343     BOOL                 sclk_polarity;         /**< SCLK_POL       */
344     SPI_CSB_ENUM         chip_select_behavior;  /**< CSB[2:0]       */
345     uint8_t              end_cycle_delay;       /**< ECD[7:0]       */
346     uint16_t             half_clock_interval;   /**< HCI[11:0]      */
347
348     double               clock_hz;              /**< if nonzero, code will compute half_clock_interval */
349     double               end_delay_ns;          /**< if nonzero, code will compute end_cycle_dealy     */
350 };
351 /*****************************************************************/
352 struct spi_status
353 {
354     BOOL     tx_status;  /* meaning this is specific to TX status, not RX status */
355     BOOL     full;
356     BOOL     empty;
357     int      fifo_size;
358     int      fifo_count;
359 };
360
361
362 /*****************************************************************/
363 #define IDI_DIN_GROUP_SIZE  8
364 #define IDI_DIN_SHIFT_RIGHT 3
365 #define IDI_DIN_GROUP_QTY   6
366 #define IDI_DIN_QTY         ( IDI_DIN_GROUP_SIZE * IDI_DIN_GROUP_QTY )
367
368
369 struct din_cfg
370 {
```

```
    struct
    {
        BOOL    falling_edge;
        BOOL    interrupt_enable;
    } chan[IDI_DIN_QTY];
};



/*************************************************************************/
#define FRAM_BLOCK_SIZE      256
#define IDI_MESSAGE_SIZE     256
#define SPI_BLOCK_SIZE       256

struct idi_dataset
{
    struct din_cfg      din_cfg;
    struct spi_cfg      spi_cfg;
    IDI_BANK_ENUM       bank_previous;
    uint16_t            base_address;
    unsigned int        irq_number;
    uint16_t            spi_id;
    BOOL                io_simulate;
    BOOL                io_report;
    const char *        svn_revision_string;
    /* Passing messages from main() to the final function which wishes to pick up
     * and install an interrupt handler can do so.
     */
    BOOL                irq_please_install_handler_request;
    BOOL                irq_handler_active;
    size_t              irq_quantity;
    volatile size_t     irq_count;
    size_t              irq_count_previous;

    BOOL                loop_command;

    /* this is a scratch pad for FRAM transactions */
    uint8_t             fram_block[FRAM_BLOCK_SIZE];

    /* this is a scratch pad for SPI transactions */
    uint8_t             spi_block[SPI_BLOCK_SIZE];

    /* this is a scratch pad area for report generation */
    char *              message[IDI_MESSAGE_SIZE];
};



/*****************************************************************************************************//**
 * @ingroup idi
 * @brief
 * Global variables
 */

/*** Software revision as determined by subversion commits */
static const char idi_svn_revision_string[] = { IDI_REV };

/*** Error code to human readable data structure */
const struct ec_human_readable ec_human_readable[] =
{
        IDI_ERROR_CODES( EC_EXTRACT_HUMAN_READABLE )
        EC_HUMAN_READABLE_TERMINATE
};

/*** a read only list of bank names - same as enumerated values */
static const char * idi_bank_symbol_names[] =
{
    "IDI_BANK_0",
    "IDI_BANK_1",
    "IDI_BANK_2",
    "IDI_BANK_3",
    "IDI_BANK_4",
    "IDI_BANK_5",
    "IDI_BANK_6",
    "IDI_BANK_7",
    "IDI_BANK_NONE",
    "IDI_BANK_UNDEFINED"
};

/*** a read only list of register parameters */
static const struct reg_definition definitions[] =
{
        IDI_REGISTER_SET_DEFINITION( REG_EXTRACT_DEFINITION )
};

/*** Global data structure which is restored during initialization and saved
 *   during application termination.
 */
struct idi_dataset  idi_dataset;
```

```
/*****************************************************************************************************/
/*****************************************************************************************************/
/*****************************************************************************************************/
/*    < < < <  R E G I S T E R  R E P O R T I N G  A N D  B A N K I N G  F U N C T I O N S  > > > >     */

/*****************************************************************//**
 * @ingroup idi
 * @brief
 * Outputs a human readable CSV to the desired output file or stdout.
 *
 * @param[in] bank  the bank enumerated value written to the bank register.
 * @return a string that describes the selected bank.
 */
static const char * IDI_Symbol_Name_Bank( IDI_BANK_ENUM bank )
{
    int index;

    switch( bank )
    {
        case IDI_BANK_0:            index = 0;  break;
        case IDI_BANK_1:            index = 1;  break;
        case IDI_BANK_2:            index = 2;  break;
        case IDI_BANK_3:            index = 3;  break;
        case IDI_BANK_4:            index = 4;  break;
        case IDI_BANK_5:            index = 5;  break;
        case IDI_BANK_6:            index = 6;  break;
        case IDI_BANK_7:            index = 7;  break;
        case IDI_BANK_NONE:         index = 8;  break;
        case IDI_BANK_UNDEFINED:    index = 9;  break;
    }
    return idi_bank_symbol_names[index];
}
/*****************************************************************//**
 * @ingroup idi
 * @brief
 * Outputs a human readable CSV to the desired output file or stdout.
 *
 * @param[in] table  register definition table or data structure array
 * @param[in] out    destination of the human readable text to specified file or terminal.
 * @return SUCCESS (0) if no errors encountered, otherwise errors are reported
 * as a negative value.
 */
int IDI_Register_Report_CSV( const struct reg_definition * table, FILE * out )
{
    int index = 0;

    fprintf( out, "\"acronym\",\"symbol\",\"bank\",\"direction\",\"physical_offset\"\n" );
    do
    {
        fprintf( out, "\"%s\",", table[index].acronym     );
        fprintf( out, "\"%s\",", table[index].symbol_name );

        switch( table[index].bank )
        {
        case IDI_BANK_0:            fprintf( out, "IDI_BANK_0" );           break;
        case IDI_BANK_1:            fprintf( out, "IDI_BANK_1" );           break;
        case IDI_BANK_2:            fprintf( out, "IDI_BANK_2" );           break;
        case IDI_BANK_3:            fprintf( out, "IDI_BANK_3" );           break;
        case IDI_BANK_4:            fprintf( out, "IDI_BANK_4" );           break;
        case IDI_BANK_5:            fprintf( out, "IDI_BANK_5" );           break;
        case IDI_BANK_6:            fprintf( out, "IDI_BANK_6" );           break;
        case IDI_BANK_7:            fprintf( out, "IDI_BANK_7" );           break;
        case IDI_BANK_NONE:         fprintf( out, "IDI_BANK_NONE" );        break;
        case IDI_BANK_UNDEFINED:    fprintf( out, "IDI_BANK_UNDEFINED" );   break;
        }
        fprintf( out, "," );

        switch( table[index].direction )
        {
            case REG_DIR_NONE:          fprintf( out, "REG_DIR_NONE" );         break;
            case REG_DIR_READ:          fprintf( out, "REG_DIR_READ" );         break;
            case REG_DIR_WRITE:         fprintf( out, "REG_DIR_WRITE" );        break;
            case REG_DIR_READ_WRITE:    fprintf( out, "REG_DIR_READ_WRITE" );   break;
        }
        fprintf( out, "," );

        fprintf( out, "\"%d\"", table[index].physical_offset );
        fprintf( out, "\n\r" ); /* separate so we have flexibility to re-organize columns */
        index++;
    } while ( definitions[index].direction != REG_DIR_NONE );

    return SUCCESS;
}

/*****************************************************************************************************/
/*****************************************************************************************************/
/*****************************************************************************************************/
/*    < < < <  M I S C E L L A N E O U S  F U N C T I O N S  > > > >                                  */


/*****************************************************************************************************//**
 * @ingroup idi
 * @brief
```

```
 * General function used to convert a string into a boolean equivalent value.
 *
 * @param[in] str  string input for conversion.
 * @return a BOOL is returned.  The default value returned is false.
 */
BOOL String_To_Bool( const char * str )
{
    switch( str[0] )
    {
    case '0':
    case 'f':
    case 'F':
        return false;
    case '1':
    case 't':
    case 'T':
        return true;
    }
    return false;
}
/*****************************************************************************************************//**
 * @ingroup idi
 * @brief
 * Dumps a hexadecimal and ASCII equivalent string to the desired output.
 * The format, illustrated below is a classic memory dump format.
 *
 * <address>: <hex> [<hex>].... <ascii list>
 *
 * @param[in] address  starting address.
 * @param[in] count    number of bytes in the buffer
 * @param[in] buffer   buffer containing the bytes to output
 * @param[out] out     desired output destination.
 * @return An error code is returned.
 */
enum { HEX_DUMP_BYTES_PER_LINE = 16 };

int Hex_Dump_Line( uint16_t address, size_t count, uint8_t * buffer, FILE * out )
{
    size_t  index;
    char    str_temp[8];
    char    str_ascii[20];
    char    str_hex_list[64];

    //if ( count > 16 ) return -EC_BUFFER_TOO_LARGE;
    if ( count > HEX_DUMP_BYTES_PER_LINE ) return -EC_HEX_DUMP_COUNT;

    sprintf( str_hex_list, "%04X: ", ((int) address) );
    strcpy( str_ascii,    "" );
    for ( index = 0; index < count; index++ )
    {   /* append/build hex list */
        sprintf( str_temp, "%02X", buffer[index] );
        strcat( str_hex_list, str_temp );
        /* output spacer in the middle and end */
        if ( 0x07 == ( index & 0x07 ) ) strcat( str_hex_list, "  " );
        /* add a space after hex value and spacers */
        strcat( str_hex_list, " " );
        /* append/build ASCII list */
        if ( ( buffer[index] < ' ' ) || ( buffer[index] > '~' ) )
        { /* since these characters will not display replace them with a period */
            strcat( str_ascii, "." );
        }
        else
        { /* print the character as is */
            sprintf( str_temp, "%c", buffer[index] );
            strcat( str_ascii, str_temp );
        }
    }
    /* compute any remaining filler required to properly align ASCII portion */
    index = strlen( str_hex_list );
    /* total = ( 6 ) 'address characters' + ( 16 * 3 ) 'hex characters' + ( 2 * 3 ) 'spacers'  = 60 */
    count = 60 - index;
    while ( count > 0 )
    { /* add sufficient characters so that the ASCII portion is in the proper columns */
        strcat( str_hex_list, " " );
        count--;
    }
    /* output the results */
    fprintf( out, "%s%s\n", str_hex_list, str_ascii );
    return SUCCESS;
}
/*****************************************************************************************************//**
 * @ingroup idi
 * @brief
 * Translates an error code into a human readable message.
 *
 * Excerpt from AES advanced Linux library/driver.
 * COPYRIGHT NOTICE Copyright (c) 2012 by Apex Embedded Systems.
 *
 * @param[in] error_code   The error code to be translated into a human readable message
 * @return a human readable string representing a very brief description of the error code.
 */
static const char ec_unknown[] = "unknown error code";

```

```
650 const char * EC_Code_To_Human_Readable( EC_ENUM error_code )
651 {
652     int index;
653
654     if ( error_code < 0 ) error_code = -error_code;
655
656     index = 0;
657     while( NULL != ec_human_readable[index].message )
658     {
659         if ( ((EC_ENUM) error_code) == ec_human_readable[index].error_code )
660         {
661             return ec_human_readable[index].message;
662         }
663         index++;
664     }
665     return ec_unknown;
666 }
667 /*****************************************************************************************//**
668  * @ingroup idi
669  * @brief
670  * Obtains a key from the keyboard in a non-blocking way.  This is exerpetted from
671  * AES Universal Library/Driver.
672  *
673  * Excerpt from AES advanced Linux library/driver.
674  * COPYRIGHT NOTICE Copyright (c) 2012 by Apex Embedded Systems.
675  *
676  * @param[out] character   Character from keyboard otherwise null character.
677  * @return If true, then a valid character is available at the keyboard, otherwise false.
678  */
679 BOOL Character_Get( int * character )
680 {
681     int  char_temp;
682     BOOL result = false;
683 #ifdef __BORLANDC__
684     if ( kbhit() )
685     {
686         char_temp = getch();
687         result    = true;
688     }
689     else
690     {
691         char_temp = '\0';
692     }
693 #else
694     char_temp = '\0';
695     result    = false;
696 #endif
697     if ( NULL != character ) *character = char_temp;
698     return result;
699 }
700
701 /*****************************************************************************************/
702 /*****************************************************************************************/
703 /*****************************************************************************************/
704 /*                 < < < <  I N T E R R U P T  H A N D L I N G   > > > >                  */
705 /*
706  *
707  * Excerpt from AES advanced library/driver for DOS with a Linux flavor.
708  *
709  * Admittedly, a whole lot of stuff for just a simple interrupt.  But it offers up
710  * many options for this board as well as the others.
711  *
712  * COPYRIGHT NOTICE Copyright (c) 2012 by Apex Embedded Systems.
713 */
714
715 #if defined( __MSDOS__ )
716
717 #include <limits.h>   /* UINT_MAX */
718
719 #define TARGET_CPU_INTEL_386 1
720
721 /*****************************************************************************/
722 /* These are the port addresses of the 8259 Programmable
723    Interrupt Controller (PIC).
724 */
725 #define IOKERN_DOS_PIC1_BASE_ADDRESS    0x20
726 #define IOKERN_DOS_PIC2_BASE_ADDRESS    0xA0
727 #define IOKERN_DOS_PIC1_CMD             IOKERN_DOS_PIC1_BASE_ADDRESS            /* PIC1 Command Port        */
728 #define IOKERN_DOS_PIC1_IMR             ( IOKERN_DOS_PIC1_BASE_ADDRESS + 1 )    /* PIC1 interrupt mask port */
729 #define IOKERN_DOS_PIC2_CMD             IOKERN_DOS_PIC2_BASE_ADDRESS            /* PIC2 Command Port        */
730 #define IOKERN_DOS_PIC2_IMR             ( IOKERN_DOS_PIC2_BASE_ADDRESS + 1 )    /* PIC2 interrupt mask port */
731
732 /* An end of interrupt needs to be sent to the Control Port of
733     the 8259 when a hardware interrupt ends. */
734 #define IOKERN_DOS_NSEOI           0x20   /* End Of Interrupt */
735
736
737 #define IOKERN_IRQ_START  disable()
738 #define IOKERN_IRQ_ENABLE enable()
739 #define IOKERN_IRQ_END(irq) IOKern_PIC_EOI( irq ); enable()
740
741 #if defined( TARGET_CPU_INTEL_186 )
742 # pragma message "TARGET_CPU_INTEL_186 for interrupt chaining"
```

```
743
744 struct pt_regs
745 {
746     uint16_t bp;
747     uint16_t di;
748     uint16_t si;
749     uint16_t ds;
750     uint16_t es;
751     uint16_t dx;
752     uint16_t cx;
753     uint16_t bx;
754     uint16_t ax;
755     uint16_t ip;
756     uint16_t cs;
757     uint16_t flags;
758 };
759
760 # define IOKERN_IRQ_CHAIN_SELECT(old_fp,regs)       \
761         _BX  =   regs.bx;                           \
762         _CX  =   regs.ax;                           \
763         regs.ax  =   FP_SEG((void far *)old_fp);    \
764         regs.bx  =   FP_OFF((void far *)old_fp);    \
765         _AX  =  _CX  ;                              \
766         __emit__(0x5D);                             \
767         __emit__(0x5F);                             \
768         __emit__(0x5E);                             \
769         __emit__(0x1F);                             \
770         __emit__(0x07);                             \
771         __emit__(0x5A);                             \
772         __emit__(0x59);                             \
773         __emit__(0xCB);
774
775 #elif defined( TARGET_CPU_INTEL_386 )
776 # pragma message "TARGET_CPU_INTEL_386 for interrupt chaining"
777
778 struct pt_regs
779 {
780     uint16_t bp;
781     union { uint32_t edi; uint16_t di; };
782     union { uint32_t esi; uint16_t si; };
783     uint16_t ds;
784     uint16_t es;
785     union { uint32_t edx; uint16_t dx; struct { uint8_t l, h; } d; };
786     union { uint32_t ecx; uint16_t cx; struct { uint8_t l, h; } c; };
787         union { uint32_t ebx; uint16_t bx; struct { uint8_t l, h; } b; };
788     /* aw is used to select top or bottom word -- used in ISR chaining */
789     union { uint32_t eax; uint16_t ax; struct { uint16_t l, h; } aw; struct { uint8_t l, h; } a; };
790     uint16_t ip;
791     uint16_t cs;
792     uint16_t flags;
793 };
794
795 # define IOKERN_IRQ_CHAIN_SELECT(old_fp,regs)   \
796     _ECX  =   regs.eax;                         \
797     regs.aw.h  = FP_SEG((void far *)old_fp);    \
798     regs.aw.l  = FP_OFF((void far *)old_fp);    \
799     _EAX  =  _ECX ;                             \
800     __emit__(0x5D);                             \
801     __emit__(0x66);  __emit__(0x5F);            \
802     __emit__(0x66);  __emit__(0x5E);            \
803     __emit__(0x1F);                             \
804     __emit__(0x07);                             \
805     __emit__(0x66);  __emit__(0x5A);            \
806     __emit__(0x66);  __emit__(0x59);            \
807     __emit__(0x66);  __emit__(0x5B);            \
808     __emit__(0xCB);
809 #endif
810
811 #define INTERRUPT interrupt
812 typedef void INTERRUPT ( * IOKERN_ISR_FP )( struct pt_regs r );
813
814 /*************************************************************************/
815 /* NOTE: Borland 5 requires semicolons - GCC will need something different */
816 #define IOKERN_LOCAL_IRQ_SAVE(flags)    \
817     do {                                \
818             asm push bx             ;   \
819             asm pushf               ;   \
820             asm pop bx              ;   \
821             asm mov [flags], bx     ;   \
822             asm cli                 ;   \
823             asm pop bx              ;   \
824     } while(0)
825
826 #define IOKERN_LOCAL_IRQ_RESTORE(flags) \
827     do {                                \
828             asm push bx             ;   \
829             asm mov bx, [flags]     ;   \
830             asm push bx             ;   \
831             asm popf                ;   \
832             asm pop bx              ;   \
833     } while(0)
834
835 /*****************************************************/
```

```
836 typedef enum
837 {
838     IOKERN_IRQ_NONE         = UINT_MAX,
839     /* hardware interrupts */
840     IOKERN_IRQ_0            =  0, /*         timer                                                  */
841     IOKERN_IRQ_1            =  1, /*         keyboard or PS/2                                       */
842     IOKERN_IRQ_2            =  2, /*         reserved for 2nd 8259 (IRQ8-15) or IRQ2 for XT         */
843     IOKERN_IRQ_3            =  3, /* pc104 - typically COM2,4                                       */
844     IOKERN_IRQ_4            =  4, /* pc104 - typically COM1,3                                       */
845     IOKERN_IRQ_5            =  5, /* pc104 - typically LPT2 or sound                                */
846     IOKERN_IRQ_6            =  6, /* pc104 - typically floppy                                       */
847     IOKERN_IRQ_7            =  7, /* pc104 - typically LPT1                                         */
848     IOKERN_IRQ_8            =  8, /*         real-time clock                                        */
849     IOKERN_IRQ_9            =  9,
850     IOKERN_IRQ_10           = 10, /* pc104 -                                                        */
851     IOKERN_IRQ_11           = 11, /* pc104 -                                                        */
852     IOKERN_IRQ_12           = 12, /* pc104 - typically, PS2 mouse                                   */
853     IOKERN_IRQ_13           = 13, /*         Numerical processor unit                               */
854     IOKERN_IRQ_14           = 14, /* pc104 - primary IDE                                            */
855     IOKERN_IRQ_15           = 15, /* pc104 - secondary IDE                                          */
856     /* software interrupts */
857     IOKERN_INT_33           = 0x33
858 } IOKERN_IRQ_ENUM;
859
860 /*****************************************************/
861 typedef enum
862 {
863     IOKERN_TASK_ID_NONE         = -1,
864     IOKERN_TASK_ID_IRQ0         =  0,
865     IOKERN_TASK_ID_IRQ1         =  1,
866     IOKERN_TASK_ID_IRQ2         =  2,
867     IOKERN_TASK_ID_IRQ3         =  3,
868     IOKERN_TASK_ID_IRQ4         =  4,
869     IOKERN_TASK_ID_IRQ5         =  5,
870     IOKERN_TASK_ID_IRQ6         =  6,
871     IOKERN_TASK_ID_IRQ7         =  7,
872     IOKERN_TASK_ID_IRQ8         =  8,
873     IOKERN_TASK_ID_IRQ9         =  9,
874     IOKERN_TASK_ID_IRQ10        = 10,
875     IOKERN_TASK_ID_IRQ11        = 11,
876     IOKERN_TASK_ID_IRQ12        = 12,
877     IOKERN_TASK_ID_IRQ13        = 13,
878     IOKERN_TASK_ID_IRQ14        = 14,
879     IOKERN_TASK_ID_IRQ15        = 15,
880     IOKERN_TASK_ID_INT33        = 16  /* mouse function calls */
881 } IOKERN_TASK_ID_ENUM;
882
883 /*****************************************************/
884 typedef enum
885 {
886     IOKERN_CHAIN_TO_OLD_OFF     = 0,    /* if help_fp==NULL then this is always the case */
887     IOKERN_CHAIN_TO_OLD_TIMER   = 1,    /* chain to the old timer, but only when
888                                            iokern_timer_periodic_chain_count reaches
889                                            zero.
890                                         */
891     IOKERN_CHAIN_TO_OLD_NORMAL  = 2     /* chain to old interrupt */
892 } IOKERN_CHAIN_TO_OLD_ENUM;
893 /*****************************************************
894 Description
895 The following code is directly from irqreturn.h from the Linux kernel tree.
896 This is being used by the simulator and DOS to emulate Linux driver behavior
897 and specifically resource allocations.
898 */
899 /**
900  * enum irqreturn
901  * @IRQ_NONE        interrupt was not from this device
902  * @IRQ_HANDLED     interrupt was handled by this device
903  * @IRQ_WAKE_THREAD handler requests to wake the handler thread
904  */
905 enum irqreturn {
906     IRQ_NONE,
907     IRQ_HANDLED,
908     IRQ_WAKE_THREAD,
909 };
910 /* irqreturn_t is directly from Linux kernel */
911 typedef enum irqreturn irqreturn_t;
912
913 /*****************************************************/
914 typedef irqreturn_t ( * IOKERN_TASK_FP )( int irq, void * dev_id, struct pt_regs * regs );
915 typedef void        ( * IOKERN_HELP_FP )( IOKERN_TASK_ID_ENUM id, IOKERN_IRQ_ENUM irq, struct pt_regs * regs );
916
917 /*****************************************************/
918 typedef struct IOKERN_TASK_TYPE
919 {
920     //SYS_TYPE      type;           /* SYS_TYPE__SYS_TASK_TYPE          */
921     IOKERN_TASK_FP  task_fp;        /* task to be called for the IRQ    */
922     IOKERN_HELP_FP  help_fp;        /* alternate interrupt routing -- timer functions   */
923     void *          dev_id;         /* task private data                */
924     const char *    name;           /* name of the task                 */
925     size_t          speed;          /* irq/int fast or slow (slow == push to main loop tasklet) */
926     IOKERN_CHAIN_TO_OLD_ENUM chain_to_old;  /* See enum above */
927     unsigned int    number;         /* irq/int that this task is mapped too */
928     size_t          sw_int;         /* if software interrupt, then this value is set */
```

```
    irqreturn_t     result;         /* result of the task               */
    unsigned long   count;          /* number of times the IRQ has run  */
    IOKERN_ISR_FP   old_isr;        /* store old ISR here               */
    unsigned char   old_pic_state;  /* PICK MASK copy                   */
} IOKERN_TASK_TYPE;

/****************************************************************************
*/
static void IOKern_DOS_IRQ_Mask_Set( unsigned char irq, unsigned char state )
{
    unsigned int  port;
    unsigned char value;
    if ( irq < 8 )
    {
        port = IOKERN_DOS_PIC1_IMR;
    }
    else
    {
        irq = irq - 8;
        port = IOKERN_DOS_PIC2_IMR;
    }
    if ( state ) value = inp( port ) |  ( 1 << irq ); /* turn off PIC */
    else         value = inp( port ) & ~( 1 << irq ); /* turn on PIC  */

    outp( port, value );
}
/****************************************************************************
*/
static void IOKern_DOS_IRQ_Mask_Get( unsigned char irq, unsigned char * state )
{
    unsigned int  port;
    unsigned char value;
    if ( irq < 8 )
    {
        port = IOKERN_DOS_PIC1_IMR;
    }
    else
    {
        irq = irq - 8;
        port = IOKERN_DOS_PIC2_IMR;
    }
    value = inp( port ) & ( 1 << irq );
    *state = value;
}

/****************************************************************************
*/
typedef void interrupt (* IOKERN_DOS_VECTOR_TYPE )();

/****************************************************************************
*/
IOKERN_DOS_VECTOR_TYPE IOKern_DOS_Vector_Get( unsigned num )
{
    asm push es
    asm push bx
/* set ES to zero, so it points to the interrupt vector table at 0000:0000 */
    asm xor bx,bx
    asm mov es,bx
    asm mov bx,[num]
/* x4 because each IVT entry (a far address) is 4 bytes long */
    asm shl bx,1
    asm shl bx,1
/* now ES:BX points to the vector we want; IVT[num].
Read the 32-bit vector (segment + offset) 'atomically',
without an interrupt happening in the middle of the read. */
    asm les bx,es:[bx]
/* return 32-bit far address in DX:AX */
    asm mov dx,es
    asm mov ax,bx
    asm pop bx
    asm pop es
/* it DOES return a value...even though the compiler will complain! */
}

/****************************************************************************
*/
void IOKern_DOS_Vector_Set( unsigned num, IOKERN_DOS_VECTOR_TYPE h )
{
    asm push es
    asm push bx
/* set ES to zero, so it points to the interrupt vector table at 0000:0000 */
    asm xor bx,bx
    asm mov es,bx
    asm mov bx,[num]
/* x4 because each IVT entry (a far address) is 4 bytes long */
    asm shl bx,1
    asm shl bx,1
/* I don't think 8088 or 80286 can do a 32-bit store,
so shut off interrupts and store 16 bits at a time. */
    asm pushf
    asm cli
    asm mov ax,word ptr [h]
    asm mov es:[bx],ax
```

```
1022    asm mov ax,word ptr [h+2]
1023    asm mov es:[bx+2],ax
1024    asm popf
1025    asm pop bx
1026    asm pop es
1027 }
1028 /*************************************************************************
1029 */
1030 #define IOKERN_DOS_INT_GET(num)     IOKern_DOS_Vector_Get(num)
1031 #define IOKERN_DOS_INT_SET(num, fn) IOKern_DOS_Vector_Set(num, (IOKERN_DOS_VECTOR_TYPE)fn)
1032
1033 #define IOKERN_TASK_QTY  17
1034 IOKERN_TASK_TYPE  iokern_task[IOKERN_TASK_QTY];
1035
1036 /*****************************************************************************************************//**
1037  * @ingroup idi
1038  * @brief
1039  */
1040 void IOKern_PIC_EOI( unsigned char irq )
1041 {
1042     if ( irq >= 8 ) outp( IOKERN_DOS_PIC2_CMD, IOKERN_DOS_NSEOI );
1043
1044     outp( IOKERN_DOS_PIC1_CMD, IOKERN_DOS_NSEOI );
1045 }
1046 /*****************************************************************************************************//**
1047  * @ingroup idi
1048  * @brief
1049  * Supporting 'fast' interrupts
1050  */
1051 void IOKern_Interrupt_Helper( IOKERN_TASK_ID_ENUM id, IOKERN_IRQ_ENUM irq, struct pt_regs * regs )
1052 {
1053     iokern_task[id].result =
1054         ( * iokern_task[id].task_fp )( irq, iokern_task[id].dev_id, regs );
1055
1056     iokern_task[id].count++;
1057 }
1058 /*****************************************************************************************************//**
1059  * @ingroup idi
1060  * @brief
1061  * Interrupt service routine.  It will subsequently call the appropriate user function.
1062  * These are meant to be short operations.
1063  */
1064 static void INTERRUPT IOKern_ISR0( struct pt_regs r )
1065 {
1066 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1067     IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ1, IOKERN_IRQ_0, r );
1068 #else
1069     IOKERN_IRQ_START;
1070     IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ0, IOKERN_IRQ_0, &r );
1071     IOKERN_IRQ_END( IOKERN_IRQ_0 );
1072 #endif
1073 }
1074 /*****************************************************************************************************//**
1075  * @ingroup idi
1076  * @brief
1077  * Interrupt service routine.  It will subsequently call the appropriate user function.
1078  * These are meant to be short operations.
1079  */
1080 static void INTERRUPT IOKern_ISR1( struct pt_regs r )
1081 {
1082 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1083     IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ1, IOKERN_IRQ_1, r );
1084 #else
1085     IOKERN_IRQ_START;
1086     IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ1, IOKERN_IRQ_1, &r );
1087     IOKERN_IRQ_END( IOKERN_IRQ_1 );
1088 #endif
1089 }
1090 /*****************************************************************************************************//**
1091  * @ingroup idi
1092  * @brief
1093  * Interrupt service routine.  It will subsequently call the appropriate user function.
1094  * These are meant to be short operations.
1095  */
1096 static void INTERRUPT IOKern_ISR2( struct pt_regs r )
1097 {
1098 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1099     IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ2, IOKERN_IRQ_2, r );
1100 #else
1101     IOKERN_IRQ_START;
1102     IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ2, IOKERN_IRQ_2, &r );
1103     IOKERN_IRQ_END( IOKERN_IRQ_2 );
1104 #endif
1105 }
1106 /*****************************************************************************************************//**
1107  * @ingroup idi
1108  * @brief
1109  * Interrupt service routine.  It will subsequently call the appropriate user function.
1110  * These are meant to be short operations.
1111  */
1112 static void INTERRUPT IOKern_ISR3( struct pt_regs r )
1113 {
1114 #if defined( IOKERN_ISR_USE_THESE_GUTS )
```

```
1115    IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ3, IOKERN_IRQ_3, r );
1116 #else
1117    IOKERN_IRQ_START;
1118    IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ3, IOKERN_IRQ_3, &r );
1119    IOKERN_IRQ_END( IOKERN_IRQ_3 );
1120 #endif
1121 }
1122 /*****************************************************************************************************//**
1123  * @ingroup idi
1124  * @brief
1125  * Interrupt service routine.  It will subsequently call the appropriate user function.
1126  * These are meant to be short operations.
1127  */
1128 static void INTERRUPT IOKern_ISR4( struct pt_regs r )
1129 {
1130 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1131    IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ4, IOKERN_IRQ_4, r );
1132 #else
1133    IOKERN_IRQ_START;
1134    IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ4, IOKERN_IRQ_4, &r );
1135    IOKERN_IRQ_END( IOKERN_IRQ_4 );
1136 #endif
1137 }
1138 /*****************************************************************************************************//**
1139  * @ingroup idi
1140  * @brief
1141  * Interrupt service routine.  It will subsequently call the appropriate user function.
1142  * These are meant to be short operations.
1143  */
1144 static void INTERRUPT IOKern_ISR5( struct pt_regs r )
1145 {
1146 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1147    IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ5, IOKERN_IRQ_5, r );
1148 #else
1149    IOKERN_IRQ_START;
1150    IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ5, IOKERN_IRQ_5, &r );
1151    IOKERN_IRQ_END( IOKERN_IRQ_5 );
1152 #endif
1153 }
1154 /*****************************************************************************************************//**
1155  * @ingroup idi
1156  * @brief
1157  * Interrupt service routine.  It will subsequently call the appropriate user function.
1158  * These are meant to be short operations.
1159  */
1160 static void INTERRUPT IOKern_ISR6( struct pt_regs r )
1161 {
1162 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1163    IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ6, IOKERN_IRQ_6, r );
1164 #else
1165    IOKERN_IRQ_START;
1166    IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ6, IOKERN_IRQ_6, &r );
1167    IOKERN_IRQ_END( IOKERN_IRQ_6 );
1168 #endif
1169 }
1170 /*****************************************************************************************************//**
1171  * @ingroup idi
1172  * @brief
1173  * Interrupt service routine.  It will subsequently call the appropriate user function.
1174  * These are meant to be short operations.
1175  */
1176 static void INTERRUPT IOKern_ISR7( struct pt_regs r )
1177 {
1178 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1179    IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ7, IOKERN_IRQ_7, r );
1180 #else
1181    IOKERN_IRQ_START;
1182    IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ7, IOKERN_IRQ_7, &r );
1183    IOKERN_IRQ_END( IOKERN_IRQ_7 );
1184 #endif
1185 }
1186 /*****************************************************************************************************//**
1187  * @ingroup idi
1188  * @brief
1189  * Interrupt service routine.  It will subsequently call the appropriate user function.
1190  * These are meant to be short operations.
1191  */
1192 static void INTERRUPT IOKern_ISR8( struct pt_regs r )
1193 {
1194 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1195    IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ8, IOKERN_IRQ_8, r );
1196 #else
1197    IOKERN_IRQ_START;
1198    IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ8, IOKERN_IRQ_8, &r );
1199    IOKERN_IRQ_END( IOKERN_IRQ_8 );
1200 #endif
1201 }
1202 /*****************************************************************************************************//**
1203  * @ingroup idi
1204  * @brief
1205  * Interrupt service routine.  It will subsequently call the appropriate user function.
1206  * These are meant to be short operations.
1207  */
```

```
1208 static void INTERRUPT IOKern_ISR9( struct pt_regs r )
1209 {
1210 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1211     IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ9, IOKERN_IRQ_9, r );
1212 #else
1213     IOKERN_IRQ_START;
1214     IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ9, IOKERN_IRQ_9, &r );
1215     IOKERN_IRQ_END( IOKERN_IRQ_9 );
1216 #endif
1217 }
1218 /*****************************************************************************************//**
1219  * @ingroup idi
1220  * @brief
1221  * Interrupt service routine.  It will subsequently call the appropriate user function.
1222  * These are meant to be short operations.
1223  */
1224 static void INTERRUPT IOKern_ISR10( struct pt_regs r )
1225 {
1226 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1227     IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ10, IOKERN_IRQ_10, r );
1228 #else
1229     IOKERN_IRQ_START;
1230     IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ10, IOKERN_IRQ_10, &r );
1231     IOKERN_IRQ_END( IOKERN_IRQ_10 );
1232 #endif
1233 }
1234 /*****************************************************************************************//**
1235  * @ingroup idi
1236  * @brief
1237  * Interrupt service routine.  It will subsequently call the appropriate user function.
1238  * These are meant to be short operations.
1239  */
1240 static void INTERRUPT IOKern_ISR11( struct pt_regs r )
1241 {
1242 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1243     IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ11, IOKERN_IRQ_11, r );
1244 #else
1245     IOKERN_IRQ_START;
1246     IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ11, IOKERN_IRQ_11, &r );
1247     IOKERN_IRQ_END( IOKERN_IRQ_11 );
1248 #endif
1249 }
1250 /*****************************************************************************************//**
1251  * @ingroup idi
1252  * @brief
1253  * Interrupt service routine.  It will subsequently call the appropriate user function.
1254  * These are meant to be short operations.
1255  */
1256 static void INTERRUPT IOKern_ISR12( struct pt_regs r )
1257 {
1258 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1259     IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ12, IOKERN_IRQ_12, r );
1260 #else
1261     IOKERN_IRQ_START;
1262     IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ12, IOKERN_IRQ_12, &r );
1263     IOKERN_IRQ_END( IOKERN_IRQ_12 );
1264 #endif
1265 }
1266 /*****************************************************************************************//**
1267  * @ingroup idi
1268  * @brief
1269  * Interrupt service routine.  It will subsequently call the appropriate user function.
1270  * These are meant to be short operations.
1271  */
1272 static void INTERRUPT IOKern_ISR13( struct pt_regs r )
1273 {
1274 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1275     IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ13, IOKERN_IRQ_13, r );
1276 #else
1277     IOKERN_IRQ_START;
1278     IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ13, IOKERN_IRQ_13, &r );
1279     IOKERN_IRQ_END( IOKERN_IRQ_13 );
1280 #endif
1281 }
1282 /*****************************************************************************************//**
1283  * @ingroup idi
1284  * @brief
1285  * Interrupt service routine.  It will subsequently call the appropriate user function.
1286  * These are meant to be short operations.
1287  */
1288 static void INTERRUPT IOKern_ISR14( struct pt_regs r )
1289 {
1290 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1291     IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ14, IOKERN_IRQ_14, r );
1292 #else
1293     IOKERN_IRQ_START;
1294     IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ14, IOKERN_IRQ_14, &r );
1295     IOKERN_IRQ_END( IOKERN_IRQ_14 );
1296 #endif
1297 }
1298 /*****************************************************************************************//**
1299  * @ingroup idi
1300  * @brief
```

```
1301 * Interrupt service routine.  It will subsequently call the appropriate user function.
1302 * These are meant to be short operations.
1303 */
1304 static void INTERRUPT IOKern_ISR15( struct pt_regs r )
1305 {
1306 #if defined( IOKERN_ISR_USE_THESE_GUTS )
1307     IOKERN_ISR_GUTS( IOKERN_TASK_ID_IRQ15, IOKERN_IRQ_15, r );
1308 #else
1309     IOKERN_IRQ_START;
1310     IOKern_Interrupt_Helper( IOKERN_TASK_ID_IRQ15, IOKERN_IRQ_15, &r );
1311     IOKERN_IRQ_END( IOKERN_IRQ_15 );
1312 #endif
1313 }
1314 /*****************************************************************************
1315 */
1316 IOKERN_ISR_FP iokern_isr_table[IOKERN_TASK_QTY] =
1317 {
1318     IOKern_ISR0,
1319     IOKern_ISR1,
1320     IOKern_ISR2,
1321     IOKern_ISR3,
1322     IOKern_ISR4,
1323     IOKern_ISR5,
1324     IOKern_ISR6,
1325     IOKern_ISR7,
1326     IOKern_ISR8,
1327     IOKern_ISR9,
1328     IOKern_ISR10,
1329     IOKern_ISR11,
1330     IOKern_ISR12,
1331     IOKern_ISR13,
1332     IOKern_ISR14,
1333     IOKern_ISR15,
1334     NULL
1335 };
1336 /*****************************************************************************************************//**
1337  * @ingroup idi
1338  * @brief
1339  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1340  *         code is returned.
1341  */
1342 int IOKern_IRQ_Test_Helper( unsigned int irq )
1343 {
1344     int error_code = SUCCESS;
1345 #if defined( IOKERN_CPU_REGION_MAP )
1346     int index;
1347     /* test for any CPU related conflicts */
1348     index = 0;
1349     while( SYS_TYPE__NONE != iokern_irq_map_list[index].type )
1350     {
1351         if ( irq == iokern_irq_map_list[index].irq )
1352         { /* there is a possible conflict with other hardware */
1353             return( -((int)(index + 1)) );
1354         }
1355         index++;
1356     }
1357 #endif
1358
1359     switch( (IOKERN_IRQ_ENUM) irq )
1360     {
1361         case IOKERN_IRQ_NONE:
1362         //case IOKERN_IRQ_0:   -- don't want users to use the timer tick...
1363         case IOKERN_IRQ_1:
1364         case IOKERN_IRQ_2:
1365         case IOKERN_IRQ_3:
1366         case IOKERN_IRQ_4:
1367         case IOKERN_IRQ_5:
1368         case IOKERN_IRQ_6:
1369         case IOKERN_IRQ_7:
1370         case IOKERN_IRQ_9:
1371         case IOKERN_IRQ_10:
1372         case IOKERN_IRQ_11:
1373         case IOKERN_IRQ_12:
1374         /* case IOKERN_IRQ_13: -- not allowed due NPU, not avail in PC104 */
1375         case IOKERN_IRQ_14:
1376         case IOKERN_IRQ_15:
1377         /* no longer usable case IOKERN_INT_33:  mouse function calls */
1378             break;
1379         default:
1380             error_code =  -EC_INTR_ERROR;
1381     }
1382
1383     return( error_code );
1384 }
1385 /*****************************************************************************************************//**
1386  * @ingroup idi
1387  * @brief
1388  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1389  *         code is returned.
1390  */
1391 int IOKern_IRQ_Test( unsigned int irq )
1392 {
1393     int error_code;
```

```
1394
1395    error_code = IOKern_IRQ_Test_Helper( irq );
1396/*  TODO:  post error to error handler
1397    DEBUG: test error_code here to determine exact cause */
1398    if ( error_code ) return( -EC_INTR_ERROR );
1399
1400    return( SUCCESS );
1401}
1402/*****************************************************************************************//**
1403 * @ingroup idi
1404 * @brief
1405 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1406 *         code is returned.
1407 */
1408void IOKern_IRQ_Invoke_ISR_Test( size_t irq )
1409{
1410    struct pt_regs r;
1411
1412    memset( &r, 0, sizeof( struct pt_regs ) );
1413    ( * iokern_isr_table[irq] )( r );
1414}
1415/*****************************************************************************************//**
1416 * @ingroup idi
1417 * @brief
1418 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1419 *         code is returned.
1420 */
1421IOKERN_TASK_ID_ENUM IOKern_Task_ID_Get( unsigned int irq )
1422{
1423    IOKERN_TASK_ID_ENUM  task_id;
1424
1425    if ( IOKERN_IRQ_NONE == irq )
1426    {
1427        task_id = IOKERN_TASK_ID_NONE;  /* nothing to do */
1428    }
1429    else if ( irq <= IOKERN_IRQ_15 )
1430    {
1431        task_id = (IOKERN_TASK_ID_ENUM) irq;
1432    }
1433    else
1434    {
1435        task_id = IOKERN_TASK_ID_NONE;
1436    }
1437    return task_id;
1438}
1439/*****************************************************************************************//**
1440 * @ingroup idi
1441 * @brief
1442 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1443 *         code is returned.
1444 */
1445IOKERN_TASK_TYPE * IOKern_Task_Handle_Get( unsigned int irq )
1446{
1447    size_t  task_id;
1448
1449    if ( SUCCESS != IOKern_IRQ_Test( irq ) ) return( NULL );
1450
1451    task_id = 0;
1452    while( task_id < IOKERN_TASK_QTY )
1453    {
1454        if ( irq == iokern_task[task_id].number )
1455        {
1456            return( &iokern_task[task_id] );
1457        }
1458        task_id++;
1459    }
1460    /* not found */
1461    return( NULL );
1462}
1463/*****************************************************************************************//**
1464 * @ingroup idi
1465 * @brief
1466 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1467 *         code is returned.
1468 */
1469static int IOKern_DOS_ISR_Install( unsigned char irq )
1470{
1471    IOKERN_TASK_ID_ENUM  task_id;
1472    int                  int_number;
1473
1474    task_id = IOKern_Task_ID_Get( irq );
1475    if ( IOKERN_TASK_ID_NONE == task_id ) return SUCCESS; /* nothing to install */
1476
1477    if ( ( IOKERN_TASK_ID_IRQ0 == task_id ) || ( IOKERN_TASK_ID_INT33 == task_id ) )
1478    { /* avoid timer interrupt and mouse stuff for now */
1479        return -EC_INTERRUPT_UNAVAILABLE;
1480    }
1481
1482//  if ( IOKERN_INT_33 == irq )
1483//  {
1484//      int_number = irq;
1485//  //  disable();
1486//      iokern_task[task_id].old_isr = (IOKERN_ISR_FP) IOKERN_DOS_INT_GET( int_number );
```

```
1487 //      //IOKern_DOS_IRQ_Mask_Get( irq, &(iokern_task[task_id].old_pic_state) );
1488 //      IOKERN_DOS_INT_SET( int_number, iokern_isr_table[task_id] );
1489 //      //IOKern_DOS_IRQ_Mask_Set( irq, 0 );  /* turn on PIC */
1490 //  //  enable();
1491 //  }
1492 //  else
1493 //  {
1494         if ( irq < 8 ) int_number = irq + 0x08;
1495         else           int_number = irq + 0x70;
1496
1497     //  disable();
1498         iokern_task[task_id].old_isr = (IOKERN_ISR_FP) IOKERN_DOS_INT_GET( int_number );
1499         IOKern_DOS_IRQ_Mask_Get( irq, &(iokern_task[task_id].old_pic_state) );
1500         IOKERN_DOS_INT_SET( int_number, iokern_isr_table[task_id] );
1501         IOKern_DOS_IRQ_Mask_Set( irq, 0 );  /* turn on PIC */
1502     //  enable();
1503 //  }
1504     return SUCCESS;
1505 }
1506 /*****************************************************************************************************//**
1507  * @ingroup idi
1508  * @brief
1509  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1510  *         code is returned.
1511  */
1512 static void IOKern_DOS_ISR_Restore( unsigned char irq )
1513 {
1514     IOKERN_TASK_ID_ENUM  task_id;
1515     int                  int_number;
1516
1517     task_id = IOKern_Task_ID_Get( irq );
1518     if ( IOKERN_TASK_ID_NONE == task_id ) return; /* nothing to install */
1519
1520 //  if ( IOKERN_INT_33 == irq )
1521 //  {
1522 //      int_number = irq;
1523 //      //IOKern_DOS_IRQ_Mask_Set( irq, iokern_task[task_id].old_pic_state );
1524 //      IOKERN_DOS_INT_SET( int_number, iokern_task[task_id].old_isr );
1525 //  }
1526 //  else
1527 //  {
1528         if ( irq < 8 ) int_number = irq + 0x08;
1529         else           int_number = irq + 0x70;
1530
1531         IOKern_DOS_IRQ_Mask_Set( irq, iokern_task[task_id].old_pic_state );
1532         IOKERN_DOS_INT_SET( int_number, iokern_task[task_id].old_isr );
1533 //  }
1534 }
1535 /*****************************************************************************************************//**
1536  * @ingroup idi
1537  * @brief
1538  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1539  *         code is returned.
1540  */
1541 int IOKern_DOS_IRQ_Request( unsigned int    irq,
1542                             IOKERN_TASK_FP  handler
1543 //                          unsigned long   flags,
1544 //                          const char *    dev_name,
1545 //                          void *          dev_id
1546                             )
1547 {
1548     int                     error_code;
1549     IOKERN_TASK_ID_ENUM     task_id;
1550 #define IOKERN_FUNCTION_NAME "IOKern_DOS_IRQ_Request"
1551  #if defined ( MSDOS_PRINTF_DEBUG_MESSAGES )
1552     Printf( "%s: ", IOKERN_FUNCTION_NAME );
1553     Printf( "irq=%d, ", irq );
1554  #endif
1555 #undef IOKERN_FUNCTION_NAME
1556     if ( SUCCESS != IOKern_IRQ_Test( irq ) ) return -EC_INTR_ERROR; /* invalid irq */
1557
1558     task_id = IOKern_Task_ID_Get( irq );
1559     if ( IOKERN_TASK_ID_NONE == task_id ) return SUCCESS; /* nothing to install */
1560
1561     if ( NULL == handler ) return -EC_INTR_ERROR;                       /* cannot have a null handler */
1562     if ( NULL != iokern_task[task_id].task_fp ) return -EC_BUSY;    /* already used */
1563 //  iokern_task[task_id].type       = SYS_TYPE_MAKE(IOKERN_TASK_TYPE);
1564     iokern_task[task_id].task_fp    = handler;
1565
1566 //  if ( IRQF_TIMER & flags )
1567 //  {
1568 //      iokern_task[task_id].help_fp = IOKern_DOS_Timer_Periodic_Interrupt;
1569 //      iokern_task[task_id].chain_to_old = IOKERN_CHAIN_TO_OLD_OFF;
1570 //      switch( flags & IOKERN_CHAIN_TO_OLD__MASK )
1571 //      {
1572 //          case IOKERN_CHAIN_TO_OLD_OFF:
1573 //              break;
1574 //          case IOKERN_CHAIN_TO_OLD_TIMER:
1575 //              if ( IOKERN_DOS_TIMER_PERIODIC_HW_IRQ == irq ) iokern_task[task_id].chain_to_old = IOKERN_CHAIN_TO_OLD_TIMER;
1576 //              break;
1577 //          case IOKERN_CHAIN_TO_OLD_NORMAL:
1578 //              //if ( IOKERN_DOS_TIMER_PERIODIC_HW_IRQ != irq ) iokern_task[task_id].chain_to_old =
     IOKERN_CHAIN_TO_OLD_NORMAL;
```

```
1579 //              break;
1580 //      }
1581 //  }
1582 //  else
1583 //  {
1584         iokern_task[task_id].help_fp        = NULL;
1585         iokern_task[task_id].chain_to_old   = IOKERN_CHAIN_TO_OLD_OFF;
1586 //  }
1587
1588 //  iokern_task[task_id].dev_id     = dev_id;
1589 //  iokern_task[task_id].name       = dev_name;
1590     iokern_task[task_id].number     = irq;
1591     error_code = IOKern_DOS_ISR_Install( (unsigned char) irq );
1592 #if defined ( MSDOS_PRINTF_DEBUG_MESSAGES )
1593     Printf( "done\n" );
1594 #endif
1595     return error_code;
1596 #undef IOKERN_FUNCTION_NAME
1597 }
1598 /*****************************************************************************************//**
1599  * @ingroup idi
1600  * @brief
1601  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1602  *         code is returned.
1603  */
1604 void IOKern_DOS_IRQ_Free( unsigned int  irq
1605 //                        void *        dev_id
1606                         )
1607 {
1608     IOKERN_TASK_TYPE *  task;
1609
1610 #define IOKERN_FUNCTION_NAME "IOKern_DOS_IRQ_Free"
1611 #if defined ( MSDOS_PRINTF_DEBUG_MESSAGES )
1612     Printf( "%s: ", IOKERN_FUNCTION_NAME );
1613     Printf( "irq=%d, ", irq );
1614 #endif
1615     task = IOKern_Task_Handle_Get( irq );
1616     if ( ( SUCCESS == IOKern_IRQ_Test( irq ) ) && ( NULL != task ) )
1617     {
1618         if ( task->task_fp )
1619         {
1620             IOKern_DOS_ISR_Restore( (unsigned char) irq );
1621             memset( task, 0, sizeof( IOKERN_TASK_TYPE ) );
1622             task->number = IOKERN_IRQ_NONE;
1623 #if defined ( MSDOS_PRINTF_DEBUG_MESSAGES )
1624             Printf( "success " );
1625 #endif
1626         }
1627     }
1628     else
1629     {
1630 #if defined ( MSDOS_PRINTF_DEBUG_MESSAGES )
1631     Printf( " " );
1632 #endif
1633     }
1634 #if defined ( MSDOS_PRINTF_DEBUG_MESSAGES )
1635     Printf( "done\n" );
1636 #endif
1637 #undef IOKERN_FUNCTION_NAME
1638 }
1639 /*****************************************************************************
1640 */
1641 void IOKern_Resource_Termination( void )
1642 {
1643     size_t              index;
1644
1645     IOKERN_IRQ_START;
1646     for ( index = 0; index < IOKERN_TASK_QTY; index++ )
1647     { /* restore any interrupts that are still mapped in */
1648         IOKern_DOS_IRQ_Free( iokern_task[index].number );
1649     }
1650     IOKERN_IRQ_ENABLE;
1651 }
1652 /*****************************************************************************
1653 */
1654 void IOKern_Resource_Initialization( void )
1655 {
1656     size_t              index;
1657
1658 //  iokern_timer_periodic_chain_count = 0;
1659 //  iokern_timer_peridic_chain_load   = 0;
1660
1661 //  memset( iokern_region_list, 0, IOKERN_REGION_QTY * sizeof( IOKERN_REGION_TYPE ) );
1662     memset( iokern_task, 0, IOKERN_TASK_QTY * sizeof( IOKERN_TASK_TYPE ) );
1663
1664     for ( index = 0; index < IOKERN_TASK_QTY; index++ )
1665     {
1666         iokern_task[index].number = IOKERN_IRQ_NONE;
1667     }
1668
1669 #if defined( IOKERN_CPU_REGION_MAP )
1670     /* map region description to region list */
1671     index = 0;
```

```
1672    while( SYS_TYPE__NONE != iokern_cpu_region_list[index].type )
1673    {
1674        iokern_cpu_region_list[index].name = iokern_cpu_region_description[index];
1675        index++;
1676    }
1677    /* map irq/int description to irq/int list */
1678    index = 0;
1679    while( SYS_TYPE__NONE != iokern_irq_map_list[index].type )
1680    {
1681        iokern_irq_map_list[index].name = iokern_irq_map_description[index];
1682        index++;
1683    }
1684 #endif
1685 }
1686
1687
1688 #endif /* __MS_DOS__ */
1689
1690 /*****************************************************************************************************/
1691
1692
1693 /*****************************************************************************************************/
1694 /*****************************************************************************************************/
1695 /*****************************************************************************************************/
1696 /*    < < < <  I / O   P R O C E S S I N G   > > > >                                                 */
1697
1698
1699 #define IDI_IO_DIRECTION_TEST   1
1700
1701 #if(0)
1702 /*****************************************************************************************************//**
1703  * @ingroup idi
1704  * @brief
1705  * Translates a register enumerated symbol into a string that is the same as the enumerated
1706  * symbol used throughout this code base.
1707  *
1708  * @param[in] location   The enumerated symbol representing the register.
1709  * @return a human readable string of the symbol.
1710  */
1711 static char * IO_Get_Symbol_Name( IDI_REG_ENUM location )
1712 {
1713     int index;
1714     index = REG_LOCATION_LOGICAL_GET( location );
1715     return definitions[index].symbol_name;
1716 }
1717 /*****************************************************************************************************//**
1718  * @ingroup idi
1719  * @brief
1720  * Looks up in the register definitions list for the ports possible read/write directions.
1721  *
1722  * @param[in] location  the enumerated register symbol.  The enumerated symbol is
1723  *                      composed of offset and bank information used to determine
1724  *                      the final address information.
1725  * @param[in] direction The desired direction that is to run
1726  * @return A false is returned if the direction is valid, otherwise a true is returned
1727  */
1728 #if defined( IDI_IO_DIRECTION_TEST )
1729 static BOOL IO_Direction_IsNotValid( IDI_REG_ENUM location, REG_DIR_ENUM direction )
1730 {
1731     int index;
1732     index = REG_LOCATION_LOGICAL_GET( location );
1733     if ( direction == (definitions[index].direction & direction) ) return false;
1734     return true;
1735 }
1736 #endif
1737 #endif
1738 /*****************************************************************************************************//**
1739  * @ingroup idi
1740  * @brief
1741  * Writes uint8_t to I/O port.  Macros are used to guide the target implementation.
1742  *
1743  * @param[in] location  the enumerated register symbol.  The enumerated symbol is
1744  *                      composed of offset and bank information used to determine
1745  *                      the final address information.
1746  * @param[in] value     The data to be written out.
1747  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1748  *         code is returned.
1749  */
1750 int IO_Write_U8( IDI_REG_ENUM location, uint8_t value )
1751 {
1752     int             index;
1753     int             offset;
1754     int             channel;
1755     int             address;
1756     IDI_BANK_ENUM   bank;
1757
1758     index   = REG_LOCATION_LOGICAL_GET( location );
1759 #if defined( IDI_IO_DIRECTION_TEST )
1760     if ( !( REG_DIR_WRITE == ( definitions[index].direction & REG_DIR_WRITE ) ) )
1761     {
1762         printf( "IO_Write_U8: %s, error in direction\n", definitions[index].symbol_name );
1763         return -EC_DIRECTION;
1764     }
```

```
#endif
    bank   = definitions[index].bank;
    if ( ( IDI_BANK_NONE != bank ) && ( bank != idi_dataset.bank_previous ) )
    {   /* write to bank register only if different -- don't bother even checking it, will take too much time. */
        address = idi_dataset.base_address + definitions[REG_LOCATION_LOGICAL_GET( IDI_BANK )].physical_offset;
        idi_dataset.bank_previous = bank;
        if ( !idi_dataset.io_simulate )
        {
#if defined( __MSDOS__ )
        outportb( address, (uint8_t) bank );
#endif
        }
        if ( ( idi_dataset.io_report ) || ( idi_dataset.io_simulate ) )
        {
        printf( "IO_Write_U8: %s, address = 0x%04X, bank = %s\n", definitions[index].symbol_name, address,
    IDI_Symbol_Name_Bank( bank ) );
        }
    }
    channel = REG_LOCATION_CHANNEL_GET( location );
    offset  = definitions[index].physical_offset;
    address = idi_dataset.base_address + offset + channel;
    if ( !idi_dataset.io_simulate )
    {
#if defined( __MSDOS__ )
    outportb( address, value );
#endif
    }
    if ( ( idi_dataset.io_report ) || ( idi_dataset.io_simulate ) )
    {
    printf( "IO_Write_U8: %s, address = 0x%04X, value = 0x%02X\n", definitions[index].symbol_name, address, value );
    }
    return SUCCESS;
}
/*****************************************************************************************************//**
 * @ingroup idi
 * @brief
 * Reads uint8_t from I/O port.  Macros are used to guide the target implementation.
 *
 * @param[in] location  the enumerated register symbol.  The enumerated symbol is
 *                      composed of offset and bank information used to determine
 *                      the final address information.
 * @param[in] value     The pointer to the destination of the read uint8_t value.
 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
 *         code is returned.
 */
int IO_Read_U8( IDI_REG_ENUM location, uint8_t * value )
{
    int             index;
    int             offset;
    int             channel;
    int             address;
    IDI_BANK_ENUM   bank;

    index   = REG_LOCATION_LOGICAL_GET( location );
#if defined( IDI_IO_DIRECTION_TEST )
    if ( !( REG_DIR_READ == ( definitions[index].direction & REG_DIR_READ ) ) )
    {
        printf( "IO_Read_U8: %s, error in direction\n", definitions[index].symbol_name );
        return -EC_DIRECTION;
    }
#endif
    bank   = definitions[index].bank;
    if ( ( IDI_BANK_NONE != bank ) && ( bank != idi_dataset.bank_previous ) )
    {   /* write to bank register only if different -- don't bother even checking it, will take too much time. */
        address = idi_dataset.base_address + definitions[REG_LOCATION_LOGICAL_GET( IDI_BANK )].physical_offset;
        idi_dataset.bank_previous = bank;
        if ( !idi_dataset.io_simulate )
        {
#if defined( __MSDOS__ )
        outportb( address, (uint8_t) bank );
#endif
        }
        if ( ( idi_dataset.io_report ) || ( idi_dataset.io_simulate ) )
        {
        printf( "IO_Read_U8: %s, address = 0x%04X, bank = %s\n", definitions[index].symbol_name, address,
    IDI_Symbol_Name_Bank( bank ) );
        }
    }
    channel = REG_LOCATION_CHANNEL_GET( location );
    offset  = definitions[index].physical_offset;
    address = idi_dataset.base_address + offset + channel;
    if ( !idi_dataset.io_simulate )
    {
#if defined( __MSDOS__ )
    *value = inportb( address );
#endif
    }
    if ( ( idi_dataset.io_report ) || ( idi_dataset.io_simulate ) )
    {
    printf( "IO_Read_U8: %s, address = 0x%04X, ", definitions[index].symbol_name, address );
#if defined( __MSDOS__ )
    printf( "value = 0x%02X\n", *value );
#else
```

```
1856     printf( "value = unknown\n" );
1857 #endif
1858     }
1859     return SUCCESS;
1860 }
1861 /*****************************************************************************************************//**
1862  * @ingroup idi
1863  * @brief
1864  * Writes uint16_t to I/O ports in a uint8_t succession incrementing the offset address.
1865  * Macros are used to guide the target implementation.  In this case, bus width (which
1866  * we typcially refer to the port width, which is different than register width) is
1867  * assumed to be byte (uint8_t) wide.
1868  *
1869  * @param[in] location  the enumerated register symbol.  The enumerated symbol is
1870  *                      composed of offset and bank information used to determine
1871  *                      the final address information.
1872  * @param[in] value     The uint16_t value to be written to the I/O register
1873  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1874  *         code is returned.
1875  */
1876 void IO_Write_U16_Address_Increment( IDI_REG_ENUM location, uint16_t value )
1877 {
1878 //TODO: assumes little endian.
1879     IO_Write_U8( (IDI_REG_ENUM)(((int) location) + 0 /* channel */), (uint8_t)(   value        & 0xFF ) );
1880     IO_Write_U8( (IDI_REG_ENUM)(((int) location) + 1 /* channel */), (uint8_t)( ( value >> 8 ) & 0xFF ) );
1881 }
1882 /*****************************************************************************************************//**
1883  * @ingroup idi
1884  * @brief
1885  * Reads uint16_t from I/O ports in a uint8_t succession incrementing the offset address.
1886  * Macros are used to guide the target implementation.  In this case, bus width (which
1887  * we typcially refer to the port width, which is different than register width) is
1888  * assumed to be byte (uint8_t) wide.
1889  *
1890  * @param[in] location  the enumerated register symbol.  The enumerated symbol is
1891  *                      composed of offset and bank information used to determine
1892  *                      the final address information.
1893  * @param[in] value     The pointer to the destination of the read uint16_t value.
1894  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1895  *         code is returned.
1896  */
1897 void IO_Read_U16_Address_Increment( IDI_REG_ENUM location, uint16_t * value )
1898 {
1899 //TODO: assumes little endian.
1900     uint8_t lsb, msb;
1901     IO_Read_U8( (IDI_REG_ENUM)(((int) location) + 0 /* channel */), &lsb );
1902     IO_Read_U8( (IDI_REG_ENUM)(((int) location) + 1 /* channel */), &msb );
1903     *value = ( ((uint16_t) msb) << 8 ) | ( ((uint16_t) lsb) & 0xFF );
1904 }
1905 /*****************************************************************************************************//**
1906  * @ingroup idi
1907  * @brief
1908  * Writes uint16_t to I/O ports in a uint8_t succession to the same address location.
1909  * Macros are used to guide the target implementation.  In this case, bus width (which
1910  * we typcially refer to the port width, which is different than register width) is
1911  * assumed to be byte (uint8_t) wide.
1912  *
1913  * @param[in] location  the enumerated register symbol.  The enumerated symbol is
1914  *                      composed of offset and bank information used to determine
1915  *                      the final address information.
1916  * @param[in] value     The uint16_t value to be written to the I/O register
1917  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1918  *         code is returned.
1919  */
1920 void IO_Write_U16_Address_Fixed( IDI_REG_ENUM location, uint16_t value )
1921 {
1922 //TODO: assumes little endian.
1923     IO_Write_U8( (IDI_REG_ENUM)(((int) location) + 0), (uint8_t)(     value      & 0xFF ) );
1924     IO_Write_U8( (IDI_REG_ENUM)(((int) location) + 0), (uint8_t)( ( value >> 8 ) & 0xFF ) );
1925 }
1926 /*****************************************************************************************************//**
1927  * @ingroup idi
1928  * @brief
1929  * Reads uint16_t from I/O ports in a uint8_t succession to the same address location.
1930  * Macros are used to guide the target implementation.  In this case, bus width (which
1931  * we typcially refer to the port width, which is different than register width) is
1932  * assumed to be byte (uint8_t) wide.
1933  *
1934  * @param[in] location  the enumerated register symbol.  The enumerated symbol is
1935  *                      composed of offset and bank information used to determine
1936  *                      the final address information.
1937  * @param[in] value     The pointer to the destination of the read uint16_t value.
1938  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1939  *         code is returned.
1940  */
1941 void IO_Read_U16_Address_Fixed( IDI_REG_ENUM location, uint16_t * value )
1942 {
1943 //TODO: assumes little endian.
1944     uint8_t lsb, msb;
1945     IO_Read_U8( (IDI_REG_ENUM)(((int) location) + 0), &lsb );
1946     IO_Read_U8( (IDI_REG_ENUM)(((int) location) + 0), &msb );
1947     *value = ( ((uint16_t) msb) << 8 ) | ( ((uint16_t) lsb) & 0xFF );
1948 }
```

```
1949
1950
1951 /*****************************************************************************************************/
1952 /*****************************************************************************************************/
1953 /*****************************************************************************************************/
1954 /*    < < < <  D I G I T A L  I N P U T  F U N C T I O N S  > > > >                                   */
1955
1956 /*****************************************************************************************************//**
1957  * @ingroup idi
1958  * @brief
1959  * Obtains the DIN component (or board ID in this case) ID number.
1960  *
1961  * @param[out] id  The 16-bit ID number
1962  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
1963  *         code is returned.
1964  */
1965 int IDI_DIN_ID_Get( uint16_t * id )
1966 {
1967     uint8_t     lsb, msb;
1968
1969     IO_Read_U8( IDI_ID_LSB, &lsb );
1970     IO_Read_U8( IDI_ID_MSB, &msb );
1971     *id = ( ((uint16_t) msb) << 8 ) | ((uint16_t) lsb);
1972 #if defined( ID_ALWAYS_REPORT_AS_GOOD )
1973     *id = ID_DIN;
1974 #endif
1975     return SUCCESS;
1976 }
1977 /*****************************************************************************************************//**
1978  * @ingroup idi
1979  * @brief
1980  * Determines if the DIN component and/or board is present.  Returns true if not present
1981  * (i.e. error).
1982  *
1983  * @return A zero is returned if present, otherwise a 1 is returned.
1984  */
1985 BOOL IDI_DIN_IsNotPresent( void )
1986 {
1987     uint16_t id;
1988     IDI_DIN_ID_Get( &id );
1989     if (ID_DIN == id ) return 0;
1990     return 1;
1991 }
1992 /*****************************************************************************************************//**
1993  * @ingroup idi
1994  * @brief
1995  * Obtains and reports a single digital input channel.
1996  *
1997  * @param[in]  channel channel to be read out.
1998  * @param[out] value   Pointer to the boolean value to be set based on the digital input value
1999  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2000  *         code is returned.
2001  */
2002 static int IDI_DIN_Channel_Get( size_t channel, BOOL * value )
2003 {
2004     size_t  group;
2005     size_t  bit;
2006     uint8_t reg_value;
2007
2008     group = channel >> IDI_DIN_SHIFT_RIGHT;
2009     bit   = channel - group * IDI_DIN_GROUP_SIZE;
2010
2011     IO_Read_U8( IDI_DI_GROUP0 + group /* channel */, &reg_value );
2012
2013     if ( 0 != ( reg_value & ( 0x01 << bit ) ) ) *value = true;
2014     else                                        *value = false;
2015
2016     return SUCCESS;
2017 }
2018 /*****************************************************************************************************//**
2019  * @ingroup idi
2020  * @brief
2021  * Reads the selected digital input port (8-bits).
2022  *
2023  * @param[in]  group  the group, range is 0 to 5.
2024  * @param[out] value  pointer to the destination for the data read out
2025  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2026  *         code is returned.
2027  */
2028 static int IDI_DIN_Group_Get( size_t group, uint8_t * value )
2029 {
2030     IO_Read_U8( IDI_DI_GROUP0 + group /* channel */, value );
2031     return SUCCESS;
2032 }
2033
2034
2035
2036
2037 /*****************************************************************************************************/
2038 /*****************************************************************************************************/
2039 /*****************************************************************************************************/
2040 /*    < < < <  S P I  F U N C T I O N S  > > > >                                                     */
2041
```

```
2042 /*****************************************************************************************//**
2043  * @ingroup idi
2044  * @brief
2045  * Retrieves the SPI ID register value.
2046  *
2047  * @param[out] id   The SPI component ID value.
2048  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2049  *         code is returned.
2050  */
2051 int SPI_ID_Get( uint16_t * id )
2052 {
2053     uint8_t     lsb, msb;
2054
2055     IO_Read_U8( SPI_ID_LSB, &lsb );
2056     IO_Read_U8( SPI_ID_MSB, &msb );
2057     *id = ( ((uint16_t) msb) << 8 ) | ((uint16_t) lsb);
2058 #if defined( ID_ALWAYS_REPORT_AS_GOOD )
2059     *id = ID_SPI;
2060 #endif
2061     return SUCCESS;
2062 }
2063 /*****************************************************************************************//**
2064  * @ingroup idi
2065  * @brief
2066  * Reports if the SPI component is available within the register space by matching a known ID.
2067  * The SPI register map is only enabled within the hardware if the hardware mode is not zero
2068  * (i.e. M1 and M0 jumpers on the board provide a nonzero value).
2069  *
2070  * @return A zero is returned if the SPI component ID is not found within the register space.
2071  */
2072 int SPI_IsNotPresent( void )
2073 {
2074     uint16_t id;
2075     SPI_ID_Get( &id );
2076     if (ID_SPI == id ) return 0;
2077     return 1;
2078 }
2079 /*****************************************************************************************//**
2080  * @ingroup idi
2081  * @brief
2082  * Computes the half clock register value given a requested time interval.  It will also produce a
2083  * 'report' indicating the actual value (due to integer resolution) as well as a computed error between
2084  * requested and actual.  The error can be used to determine whether timing constraints are met.
2085  *
2086  * @param[in]  half_clock_request_sec  Request time interval in seconds.  Example: 20.0e-6 is 20uS.
2087  * @param[in]  half_clock_actual_sec   Actual computed time. If this pointer is NULL, then it is not output.
2088  * @param[out] error  Error between requested and actual.  If this pointer is NULL, then it is not output.
2089  * @param[out] hci    Half clock register value computed.  If this pointer is NULL, then it is not output.
2090  *
2091  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2092  *         code is returned.
2093  */
2094 int SPI_Calculate_Half_Clock( double     half_clock_request_sec, /* requested half clock interval           */
2095                               double *   half_clock_actual_sec,  /*  computed actual half clock interval    */
2096                               double *   error,                  /* error between actual and desired        */
2097                               uint16_t * hci                     /* computed count                          */
2098                             )
2099 {
2100     double      scratch;
2101     int         hci_temp;
2102
2103     /* spi_half_clock_interval_sec = CLOCK_PERIOD_SEC * ( 4.0 + ( (double) hci ) ) */
2104     scratch = ( half_clock_request_sec / CLOCK_PERIOD_SEC ) - 4.0;
2105     hci_temp = (int) scratch;
2106
2107     if ( ( hci_temp > 4095 ) || ( hci_temp < 0 ) ) return -EC_SPI_HALF_CLOCK_OUT_OF_RANGE;
2108
2109     /* compute actual */
2110     scratch = CLOCK_PERIOD_SEC * ( 4.0 + ((double) hci_temp) );
2111     if ( NULL != error                ) *error                = ( scratch - half_clock_request_sec ) /
     half_clock_request_sec;
2112     if ( NULL != half_clock_actual_sec ) *half_clock_actual_sec = scratch;
2113     if ( NULL != hci                  ) *hci                  = (uint16_t) hci_temp;
2114     return SUCCESS;
2115 }
2116 /*****************************************************************************************//**
2117  * @ingroup idi
2118  * @brief
2119  * Computes the SPI clock half clock register value given a requested SPI clock frequency.  It will also
2120  * produce a 'report' indicating the actual value (due to integer resolution) as well as a computed error
2121  * between requested and actual.  The eror can be used to determine whether timing constraints are met.
2122  *
2123  * @param[in]  clock_request_hz  Request clock frequency in Hertz.  Example: 1.0e6 is 1MHz.
2124  * @param[in]  clock_actual_hz   Actual computed frequency. If this pointer is NULL, then it is not output.
2125  * @param[out] error  Error between requested and actual.  If this pointer is NULL, then it is not output.
2126  * @param[out] hci    Half clock register value computed.  If this pointer is NULL, then it is not output.
2127  *
2128  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2129  *         code is returned.
2130  */
2131 int SPI_Calculate_Clock( double     clock_request_hz,   /* requested spi clock frequency           */
2132                          double *   clock_actual_hz,    /* computed actual clock frequency         */
2133                          double *   error,              /* error between actual and desired        */
```

```
2134                         uint16_t * hci                 /* computed count                          */
2135                        )
2136 {
2137     int      error_code;
2138     double   half_clock_request_sec;
2139     double   half_clock_actual_sec;
2140     double   error_internal;
2141     double   scratch;
2142     uint16_t hci_internal;
2143 
2144     half_clock_request_sec = 1.0 / ( 2.0 * clock_request_hz );
2145 
2146     error_code = SPI_Calculate_Half_Clock( half_clock_request_sec,
2147                                            &half_clock_actual_sec,
2148                                            &error_internal,
2149                                            &hci_internal
2150                                           );
2151     if ( error_code ) return error_code;
2152 
2153     /* compute actual frequency */
2154     scratch = 1.0 / ( 2.0 * half_clock_actual_sec );
2155     if ( NULL != error           ) *error           = ( scratch - clock_request_hz ) / clock_request_hz;
2156     if ( NULL != clock_actual_hz ) *clock_actual_hz = scratch;
2157     if ( NULL != hci             ) *hci             = (uint16_t) hci_internal;
2158     return SUCCESS;
2159 }
2160 /*****************************************************************************************************//**
2161  * @ingroup idi
2162  * @brief
2163  * Computes the half clock interval in seconds given the value from the half clock interval register.
2164  *
2165  * @param[in] half_clock_interval  Half clock interval register value
2166  * @return The time value as a double in units of seconds.
2167  */
2168 double SPI_Calculate_Half_Clock_Interval_Sec( uint16_t half_clock_interval /* hci */ )
2169 {
2170     double half_clock_interval_sec;
2171     half_clock_interval_sec = CLOCK_PERIOD_SEC * ( 4.0 + ((double) half_clock_interval ) );
2172     return half_clock_interval_sec;
2173 }
2174 /*****************************************************************************************************//**
2175  * @ingroup idi
2176  * @brief
2177  * Computes the time delay at the end of each byte transmitted.  It will only output the parameters
2178  * whose pointers are not NULL.
2179  *
2180  * @param[in]  spi_half_clock_interval_sec  Computed half clock interval in seconds
2181  * @param[in]  delay_request_sec            Requested time delay in seconds
2182  * @param[out] delay_actual_sec             Pointer to actual time delay computed, if not NULL.
2183  * @param[out] error                        Pointer to error value computed, if not NULL.
2184  * @param[out] ecd                          Pointer to the computed end-cycle-delay, if not NULL.
2185  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2186  *         code is returned.
2187  */
2188 int SPI_Calculate_End_Cycle_Delay( double    spi_half_clock_interval_sec, /* calculated half-clock interval     */
2189                                    double    delay_request_sec,           /* requested end-delay interval       */
2190                                    double *  delay_actual_sec,            /*  computed actual delay             */
2191                                    double *  error,                       /* error between actual and desired   */
2192                                    uint8_t * ecd                          /* computed count                     */
2193                                  )
2194 {
2195     //double        delay_between_words_sec;
2196     double      scratch;
2197     int         ecd_temp;
2198 
2199     /* delay_sec = CLOCK_PERIOD_SEC * 4 + ECD * spi_half_clock_interval_sec  */
2200     scratch = ( delay_request_sec - 4.0 * CLOCK_PERIOD_SEC ) / spi_half_clock_interval_sec;
2201     ecd_temp = (int) scratch;
2202 
2203     if ( ( ecd_temp > 255 ) || ( ecd_temp < 0 ) ) return -EC_SPI_ECD_OUT_OF_RANGE;
2204 
2205     /* compute actual */
2206     scratch = CLOCK_PERIOD_SEC * 4.0 + ((double) ecd_temp) * spi_half_clock_interval_sec;
2207     if ( NULL != error           ) *error           = ( scratch - delay_request_sec ) / delay_request_sec;
2208     if ( NULL != delay_actual_sec ) *delay_actual_sec = scratch;
2209     if ( NULL != ecd             ) *ecd             = (uint8_t) ecd_temp;
2210     return SUCCESS;
2211 }
2212 /*****************************************************************************************************//**
2213  * @ingroup idi
2214  * @brief
2215  * Extracts the chip select behavior from the SPI configuration register.
2216  *
2217  * @param[out] chip_select_behavior   pointer to the destination of the value obtained.
2218  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2219  *         code is returned.
2220  */
2221 int SPI_Configuration_Chip_Select_Behavior_Get( SPI_CSB_ENUM * chip_select_behavior )
2222 {
2223     uint8_t     scratch;
2224 
2225     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
2226 
```

```
2227    IO_Read_U8( SPI_CONFIG, &scratch );
2228
2229    *chip_select_behavior = (SPI_CSB_ENUM) ( scratch >> 4 ) & 0x07;
2230    return SUCCESS;
2231 }
2232 /*****************************************************************************************************//**
2233  * @ingroup idi
2234  * @brief
2235  * Sets the chip select behavior to the SPI configuration register.
2236  *
2237  * @param[in] chip_select_behavior  enumerated value to be written to the register.
2238  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2239  *         code is returned.
2240  */
2241 int SPI_Configuration_Chip_Select_Behavior_Set( SPI_CSB_ENUM chip_select_behavior )
2242 {
2243    uint8_t     scratch;
2244
2245    if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
2246
2247    IO_Read_U8( SPI_CONFIG, &scratch );
2248
2249    scratch &= 0x70;
2250    scratch |= (uint8_t) ( ( chip_select_behavior & 0x07 ) << 4 );
2251
2252    IO_Write_U8( SPI_CONFIG, scratch );
2253    return SUCCESS;
2254 }
2255 /*****************************************************************************************************//**
2256  * @ingroup idi
2257  * @brief
2258  * Commits the configuration data structure to the hardware.
2259  *
2260  * @param[in] cfg  The software configuration data structure to be committed to hardware
2261  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2262  *         code is returned.
2263  */
2264 int SPI_Configuration_Set( struct spi_cfg * cfg )
2265 {
2266    int         error_code;
2267    double      scratch;
2268    double      hci_sec;     /* half clock interval in seconds */
2269    uint8_t     config;
2270
2271    if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
2272
2273    config = (uint8_t) ( ( cfg->chip_select_behavior & 0x07 ) << 4 );
2274    if ( cfg->sclk_polarity ) config |= 0x01;
2275    if ( cfg->sclk_phase    ) config |= 0x02;
2276    if ( cfg->sdi_polarity  ) config |= 0x04;
2277    if ( cfg->sdo_polarity  ) config |= 0x08;
2278    if ( cfg->sdio_wrap     ) config |= 0x80;
2279
2280    IO_Write_U8( SPI_CONFIG, config );
2281
2282    if ( cfg->clock_hz > 0 )
2283    { /* compute half_clock_interval */
2284        //scratch = ( 1.0 - ( 8.0 * CLOCK_PERIOD_SEC * cfg->clock_hz ) ) / (  2.0 * CLOCK_PERIOD_SEC * cfg->clock_hz );
2285        error_code = SPI_Calculate_Clock( cfg->clock_hz, NULL, NULL, &(cfg->half_clock_interval) );
2286        if ( error_code ) return error_code;
2287    }
2288    hci_sec = SPI_Calculate_Half_Clock_Interval_Sec( cfg->half_clock_interval );
2289
2290    if ( cfg->end_delay_ns > 0 )
2291    {
2292        scratch    = cfg->end_delay_ns * 1.0e-9;
2293        error_code = SPI_Calculate_End_Cycle_Delay( hci_sec,                /* calculated half-clock interval     */
2294                                                    scratch,                /* requested end-delay interval       */
2295                                                    NULL,                   /*  computed actual delay             */
2296                                                    NULL,                   /* error between actual and desired   */
2297                                                    &(cfg->end_cycle_delay) /* computed count                     */
2298                                                   );
2299        if ( error_code ) return error_code;
2300    }
2301
2302    IO_Write_U8( SPI_HCI_LSB, (uint8_t)( cfg->half_clock_interval & 0xFF ) );
2303    IO_Write_U8( SPI_HCI_MSB, (uint8_t)( cfg->half_clock_interval >> 8   ) );
2304    IO_Write_U8( SPI_ECD, cfg->end_cycle_delay );
2305
2306    memcpy( &(idi_dataset.spi_cfg), &cfg, sizeof( struct spi_cfg ) );
2307
2308    return SUCCESS;
2309 }
2310 /*****************************************************************************************************//**
2311  * @ingroup idi
2312  * @brief
2313  * Obtains the SPI configuration from the hardware.
2314  *
2315  * @param[out] cfg  SPI configuration data structure or data set
2316  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2317  *         code is returned.
2318  */
2319 int SPI_Configuration_Get( struct spi_cfg * cfg )
```

```
2320 {
2321     uint8_t     scratch;
2322 
2323     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
2324 
2325     IO_Read_U8( SPI_CONFIG, &scratch );
2326     cfg->chip_select_behavior   = (SPI_CSB_ENUM) ( scratch >> 4 ) & 0x07;
2327     cfg->sclk_polarity          = (BOOL) ( scratch & 0x01 );
2328     cfg->sclk_phase             = (BOOL) ( scratch & 0x02 );
2329     cfg->sdi_polarity           = (BOOL) ( scratch & 0x04 );
2330     cfg->sdo_polarity           = (BOOL) ( scratch & 0x08 );
2331     cfg->sdio_wrap              = (BOOL) ( scratch & 0x80 );
2332 
2333     IO_Read_U8( SPI_HCI_LSB, &scratch );
2334     cfg->half_clock_interval = (uint16_t) scratch;
2335     IO_Read_U8( SPI_HCI_MSB, &scratch );
2336     cfg->half_clock_interval |= ( (uint16_t) scratch) << 8;
2337 
2338     IO_Read_U8( SPI_ECD, &(cfg->end_cycle_delay) );
2339 
2340     cfg->clock_hz = 1.0 / ( 2.0 * CLOCK_PERIOD_SEC * ( 4.0 + ((double) cfg->half_clock_interval) ) );
2341     cfg->end_delay_ns = 1.0e9 * CLOCK_PERIOD_SEC * 4.0 + 0.5 * ((double) cfg->end_cycle_delay) / cfg->clock_hz;
2342 
2343     memcpy( &(idi_dataset.spi_cfg), &cfg, sizeof( struct spi_cfg ) );
2344     return SUCCESS;
2345 }
2346 /*****************************************************************************************************//**
2347  * @ingroup idi
2348  * @brief
2349  * Creates a human readable report of the SPI configuration data structure.
2350  *
2351  * @param[in] cfg  SPI configuration data structure pointer
2352  * @parm[out] out  File destination descriptor
2353  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2354  *         code is returned.
2355  */
2356 int SPI_Report_Configuration_Text( struct spi_cfg * cfg, FILE * out )
2357 {
2358     fprintf( out, "################################## SPI Configuration:\n" );
2359     fprintf( out, "sdio_wrap             = %s\n", cfg->sclk_polarity    ? "true" : "false" );
2360     fprintf( out, "sdo_polarity          = %s\n", cfg->sclk_polarity    ? "true" : "false" );
2361     fprintf( out, "sdi_polarity          = %s\n", cfg->sdi_polarity     ? "true" : "false" );
2362     fprintf( out, "sclk_phase            = %s\n", cfg->sclk_phase       ? "true" : "false" );
2363     fprintf( out, "sclk_polarity         = %s\n", cfg->sclk_polarity    ? "true" : "false" );
2364     fprintf( out, "chip_select_behavior  = " );
2365 
2366     switch( cfg->chip_select_behavior )
2367     {
2368         case IDI_CSB_SOFTWARE:      fprintf( out, "IDI_CSB_SOFTWARE" );         break;
2369         case IDI_CSB_BUFFER:        fprintf( out, "IDI_CSB_BUFFER" );           break;
2370         case IDI_CSB_uint8_t:           fprintf( out, "IDI_CSB_uint8_t" );          break;
2371         case IDI_CSB_uint16_t:      fprintf( out, "IDI_CSB_uint16_t" );         break;
2372         default:                    fprintf( out, "undefined" );                break;
2373     }
2374     fprintf( out, "\n" );
2375 
2376     fprintf( out, "end_cycle_delay       = 0x%02X  (%d)\n", cfg->end_cycle_delay,     cfg->end_cycle_delay     );
2377     fprintf( out, "half_clock_interval   = 0x%04X  (%d)\n", cfg->half_clock_interval, cfg->half_clock_interval );
2378 
2379     fprintf( out, "clock_hz              = %f Hz\n", cfg->clock_hz );
2380     fprintf( out, "end_delay_ns          = %f ns\n", cfg->end_delay_ns );
2381     fprintf( out, "\n" );
2382     return SUCCESS;
2383 }
2384 /*****************************************************************************************************//**
2385  * @ingroup idi
2386  * @brief
2387  * Produces a human readable report of the SPI status data structure.
2388  *
2389  * @param[in] status SPI status data structure pointer
2390  * @parm[out] out    File destination descriptor
2391  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2392  *         code is returned.
2393  */
2394 int SPI_Report_Status_Text( struct spi_status * status, FILE * out )
2395 {
2396     fprintf( out, "################################## SPI " );
2397     if ( status->tx_status ) fprintf( out, "TX" );
2398     else                     fprintf( out, "RX" );
2399 
2400     fprintf( out, " Status:\n" );
2401 
2402     fprintf( out, "full       = %s\n", status->full  ? "true" : "false" );
2403     fprintf( out, "empty      = %s\n", status->empty ? "true" : "false" );
2404     fprintf( out, "fifo size  = %d\n", status->fifo_size                );
2405     fprintf( out, "fifo count = %d\n", status->fifo_count               );
2406 
2407     return SUCCESS;
2408 }
2409 /*****************************************************************************************************//**
2410  * @ingroup idi
2411  * @brief
2412  * Initializes the SPI configuration data structure
```

```
2413 *
2414 * @parma[in] cfg  Pointer to the SPI configuration data structure to be initialized
2415 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2416 *         code is returned.
2417 */
2418 int SPI_Configuration_Initialize( struct spi_cfg * cfg )
2419 {
2420     cfg->sdio_wrap              = false;
2421     cfg->sdo_polarity           = false;
2422     cfg->sdi_polarity           = false;
2423     /*  Mode    CPOL    CPHA
2424      *   0       0       0
2425      *   1       0       1
2426      *   2       1       0
2427      *   3       1       1
2428      */
2429     cfg->sclk_phase             = false;   /* the FRAM uses SPI Mode 0 or 3 */
2430     cfg->sclk_polarity          = false;
2431
2432     cfg->chip_select_behavior   = false;
2433     cfg->end_cycle_delay        = 0;        /* shortest delay possible      */
2434     cfg->half_clock_interval    = 0;        /* shortest interval possible   */
2435
2436     cfg->clock_hz               = 0.0;
2437     cfg->end_delay_ns           = 0.0;
2438
2439     return SUCCESS;
2440 }
2441 /*****************************************************************************************************//**
2442  * @ingroup idi
2443  * @brief
2444  * Builds a detailed status data structure of the transmit/write outgoing SPI data FIFO.
2445  * Reports the quantity of bytes currently in the transmit FIFO, full flag, empty flag,
2446  * the total size of the FIFO in bytes, and sets tx_status to true indicating that this
2447  * is status specific to the transmit FIFO.
2448  *
2449  * The status register has the following format:
2450  * status[7]    full
2451  * status[6]    empty
2452  * status[5]    not used (future size expansion)
2453  * status[4:0]  number of bytes currently in the FIFO
2454  *
2455  * @param[out] status   Pointer to status data structure to be updated.
2456  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2457  *         code is returned.
2458  */
2459 int SPI_Status_Write( struct spi_status * status )
2460 {
2461     uint8_t reg_value;
2462
2463     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
2464
2465     IO_Read_U8( SPI_TX_STATUS, &reg_value );
2466     status->fifo_count = (int)( reg_value & 0x1F );
2467     status->full       = (BOOL)( reg_value & 0x80 );
2468     status->fifo_size  = (int) SPI_FIFO_SIZE;
2469     status->empty      = (BOOL)( reg_value & 0x40 );
2470     status->tx_status  = true;
2471     return SUCCESS;
2472 }
2473 /*****************************************************************************************************//**
2474  * @ingroup idi
2475  * @brief
2476  * Returns the complete write/transmit FIFO status.
2477  *
2478  * The status register has the following format:
2479  * status[7]    full
2480  * status[6]    empty
2481  * status[5]    not used (future size expansion)
2482  * status[4:0]  number of bytes currently in the FIFO
2483  *
2484  *
2485  * @param[out] full              FIFO full flag
2486  * @param[out] empty             FIFO empty flag
2487  * @param[out] bytes_in_fifo     a count of the number of bytes in the FIFO
2488  * @return     nothing
2489  */
2490 void SPI_Status_Write_FIFO_Status( BOOL * full, BOOL * empty, size_t * bytes_in_fifo )
2491 {
2492     uint8_t reg_value;
2493     IO_Read_U8( SPI_TX_STATUS, &reg_value );
2494     switch( reg_value & 0xC0 )
2495     {
2496         case 0x00: *full = false;   *empty = false;     break;
2497         case 0x40: *full = false;   *empty = true;      break;
2498         case 0x80: *full = true;    *empty = false;     break;
2499         case 0xC0: *full = true;    *empty = true;      break;
2500     }
2501     *bytes_in_fifo = (size_t) SPI_FIFO_SIZE - (size_t)( reg_value & 0x1F );
2502 }
2503 /*****************************************************************************************************//**
2504  * @ingroup idi
2505  * @brief
```

```
 * Returns the transmit/write FIFO full status flag.   It is preferable to use the SPI_Status_Write()
 * or SPI_Status_Write_FIFO_Status() because all status is retrieved at one time.
 *
 * @return Returns true if the transmit/write FIFO is full.
 */
BOOL SPI_Status_Write_FIFO_Is_Full( void )
{
    uint8_t reg_value;
    IO_Read_U8( SPI_TX_STATUS, &reg_value );
    if ( reg_value & 0x80 ) return true;
    return false;
}
/*****************************************************************************************************//**
 * @ingroup idi
 * @brief
 * Returns the complete read/receive FIFO status.
 *
 * The status register has the following format:
 * status[7]    full
 * status[6]    empty
 * status[5]    not used (future size expansion)
 * status[4:0]  number of bytes currently in the FIFO
 *
 *
 * @param[out] empty             FIFO empty flag
 * @param[out] bytes_available   a count of the number of bytes in the FIFO
 * @return     nothing
 */
void SPI_Status_Read_FIFO_Status( BOOL * empty, size_t * bytes_available )
{
    uint8_t reg_value;

    IO_Read_U8( SPI_RX_STATUS, &reg_value );
    *bytes_available = (size_t)( reg_value & 0x1F );
    *empty           = (BOOL)( reg_value & 0x40 );
}
/*****************************************************************************************************//**
 * @ingroup idi
 * @brief
 * Builds a detailed status data structure of the receive/read incoming SPI data FIFO.
 * Reports the quantity of bytes currently in the receive FIFO, full flag, empty flag,
 * the total size of the FIFO in bytes, and sets tx_status to false indicating that this
 * is status specific to the receive FIFO.
 *
 * The status register has the following format:
 * status[7]    full
 * status[6]    empty
 * status[5]    not used (future size expansion)
 * status[4:0]  number of bytes currently in the FIFO
 *
 * @param[out] status   Pointer to status data structure to be updated.
 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
 *         code is returned.
 */
int SPI_Status_Read( struct spi_status * status )
{
    uint8_t reg_value;

    if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;

    IO_Read_U8( SPI_RX_STATUS, &reg_value );
    status->fifo_count = (int)( reg_value & 0x1F );
    status->full       = (BOOL)( reg_value & 0x80 );
    status->fifo_size  = (int) SPI_FIFO_SIZE;
    status->empty      = (BOOL)( reg_value & 0x40 );
    status->tx_status  = false;
    return SUCCESS;
}
/*****************************************************************************************************//**
 * @ingroup idi
 * @brief
 * Returns the transmit/write FIFO empty status flag.  This function is typically used to wait for the
 * transmit/write FIFO to become empty.
 *
 * @return Returns true if the transmit/write FIFO is not empty.
 */
BOOL SPI_Status_Write_FIFO_Is_Not_Empty( void )
{
    uint8_t reg_value;
    IO_Read_U8( SPI_TX_STATUS, &reg_value );
    if ( reg_value & 0x40 ) return false;
    return true;
}
/*****************************************************************************************************//**
 * @ingroup idi
 * @brief
 * Returns the receive/read FIFO empty status flag.  This function is typically used to determine
 * if the FIFO is empty.
 *
 * @return Returns true if the receive/read FIFO is not empty.
 */
BOOL SPI_Status_Read_FIFO_Is_Not_Empty( void )
{
```

```
2599     uint8_t reg_value;
2600
2601     IO_Read_U8( SPI_RX_STATUS, &reg_value );
2602     if ( reg_value & 0x40 ) return false;
2603     return true;
2604 }
2605 /**************************************************************************************************//**
2606  * @ingroup idi
2607  * @brief
2608  * Sets/Clears the chip select or used to commit the transmit/write FIFO to the spi interface.  The mode
2609  * of operation is dependent on the chip_select_behavior.
2610  *
2611  * @param[in] chip_select   Used to write to the SCS_COMMIT bit.  Its behavior is dependent on the
2612  *                          chip_select_behavior.
2613  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2614  *         code is returned.
2615  */
2616 int SPI_Commit( uint8_t chip_select )
2617 {
2618     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
2619     IO_Write_U8( SPI_COMMIT, chip_select );
2620     return SUCCESS;
2621 }
2622 /**************************************************************************************************//**
2623  * @ingroup idi
2624  * @brief
2625  * Writes specifically to the SPI transmit/write data FIFO.  It does not attempt to correlate
2626  * the number of transmit bytes with receive bytes.  Its purpose is more for low level hardware
2627  * testing.  Note that this function has a signature identical to the fwrite() function (i.e.
2628  * make use of function pointers to guide destination of data).
2629  *
2630  * @param[in] buffer   Buffer containing the data to be written.
2631  * @param[in] size     Size of objects in bytes.
2632  * @param[in] count    Number of objects to be written
2633  * @param[out] fd_log  Optional log file to write the buffer too.  If NULL, then no logging.
2634  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2635  *         code is returned indicating an error.  In addition, a positive value is returned
2636  *         indicating the number of actual objects written.
2637  */
2638 int SPI_FIFO_Write( const void * buffer, size_t size, size_t count, FILE * fd_log )
2639 {
2640     int     error_code;       /* used primarily for debug purposes */
2641     size_t  bytes_in_fifo;
2642     BOOL    empty;
2643     BOOL    full;
2644     size_t  index;
2645     size_t  qty_objects;
2646     size_t  qty_bytes;
2647
2648     error_code = SUCCESS;
2649
2650     if ( (size * count) > SPI_FIFO_SIZE ) return -EC_PARAMETER;
2651
2652     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
2653
2654     SPI_Status_Write_FIFO_Status( &full, &empty, &bytes_in_fifo );
2655
2656     qty_objects =  (SPI_FIFO_SIZE - bytes_in_fifo) / size; /* max number of objects that can be processed */
2657     if ( count > qty_objects ) count = qty_objects;
2658
2659     qty_bytes = count * size;
2660
2661     for ( index = 0; index < qty_bytes; index++ ) IO_Write_U8( SPI_DATA, ((uint8_t *) buffer)[index] );
2662
2663     if ( NULL != fd_log )
2664     {
2665         error_code = fwrite( buffer, size, qty_objects, fd_log );
2666     }
2667
2668     if ( SUCCESS == error_code ) error_code = ( (int) qty_objects );
2669     return error_code;
2670 }
2671 /**************************************************************************************************//**
2672  * @ingroup idi
2673  * @brief
2674  * Reads from the SPI receive/read data FIFO.  It does not attempt to correlate the number of
2675  * transmit bytes with receive bytes.  Its purpose is more for low level hardware
2676  * testing.  Note that this function has a signature identical to the fread() function (i.e.
2677  * make use of function pointers to guide sourcing of data).
2678  *
2679  * @param[in] buffer   Buffer for the data destination.
2680  * @param[in] size     Size of objects in bytes.
2681  * @param[in] count    Number of objects to be read
2682  * @param[out] fd_log  Optional log file to write the buffer too.  If NULL, then no logging.
2683  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2684  *         code is returned indicating an error.  In addition, a positive value is returned
2685  *         indicating the number of actual objects written.
2686  */
2687 int SPI_FIFO_Read( const void * buffer, size_t size, size_t count, FILE * fd_log )
2688 {
2689     int     error_code;       /* used primarily for debug purposes */
2690     size_t  bytes_available;
2691     BOOL    empty;
```

```
    size_t  index;
    size_t  qty_objects;
    size_t  qty_bytes;

    error_code = SUCCESS;

    if ( (size * count) > SPI_FIFO_SIZE ) return -EC_PARAMETER;

    if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;

    SPI_Status_Read_FIFO_Status( &empty, &bytes_available );

    qty_objects =  bytes_available / size; /* max number of objects that can be processed */
    if ( count > qty_objects ) count = qty_objects;

    qty_bytes = count * size;

    for ( index = 0; index < qty_bytes; index++ ) IO_Read_U8( SPI_DATA, &(((uint8_t *) buffer)[index]) );

    if ( NULL != fd_log )
    {
        error_code = fwrite( buffer, size, qty_objects, fd_log );
    }

    if ( SUCCESS == error_code ) error_code = ( (int) qty_objects );
    return error_code;
}
/*****************************************************************************************************//**
 * @ingroup idi
 * @brief
 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
 *         code is returned.
 */
static int SPI_Data_Write_Read_Helper(  size_t          size,       /**< object size: u8 = 1, u16 = 2 */
                                        size_t          tx_count,   /**< object count  */
                                        const void *    tx_buffer,
                                        size_t          rx_size,    /**< object count  */
                                        const void *    rx_buffer,
                                        BOOL            active_tx,
                                        BOOL            active_rx,
                                        SPI_CSB_ENUM    csb
                                     )
{
    size_t  rx_bytes_available;
    BOOL    rx_empty;
    BOOL    tx_full;
    BOOL    tx_empty;
    size_t  tx_bytes_available;
    size_t  tx_index;
    size_t  rx_index;
    size_t  index;
    BOOL    commit_valid;
    uint8_t bit_bucket; /* tossed */
    (void)  rx_size;

    /* verify size information */
    switch ( size )
    {
        case sizeof( uint8_t  ):
        case sizeof( uint16_t ):
            break; /* these sizes are OK */
        default:
            return -EC_SPI_OBJECT_SIZE;
            //break;
    }

    /* initially need to make sure that both TX and RX are empty */
    do
    { //TODO: need a time out of some sort here and then return an error code.
        SPI_Status_Write_FIFO_Status( &tx_full, &tx_empty, &tx_bytes_available );
        if ( false == tx_empty ) SPI_Commit( 0xFF );
    } while ( false == tx_empty );

    do
    { //TODO: need a time out of some sort here and then return an error code.
        SPI_Status_Read_FIFO_Status( &rx_empty, &rx_bytes_available );
        if ( false == rx_empty ) IO_Read_U8( SPI_DATA, &bit_bucket  ); /* toss */
    } while ( false == rx_empty );

    tx_index     = 0;
    rx_index     = 0;
    commit_valid = false;
    while ( tx_index < tx_count )
    {
        /* get status simultaneously */
        SPI_Status_Write_FIFO_Status( &tx_full, &tx_empty, &tx_bytes_available );
        if ( (true == tx_full) && (false == commit_valid) )
        {
            if ( IDI_CSB_SOFTWARE != csb ) SPI_Commit( 0xFF ); /* does not matter what is written */
            commit_valid = true;
        }
        if ( (true == tx_empty) && (true == commit_valid) )
        {
```

```
2785            commit_valid = false; /* will need to restart the buffer transmission */
2786        }
2787        /* Write Data
2788         *
2789         */
2790        if ( tx_bytes_available > size )
2791        {   /* write data */
2792            if ( active_tx )
2793            {
2794            for ( index = 0; index < size; index++ ) IO_Write_U8( SPI_DATA, ((uint8_t *) tx_buffer)[tx_index] );
2795            }
2796            else
2797            {
2798            for ( index = 0; index < size; index++ ) IO_Write_U8( SPI_DATA, 0x00 ); /* send anything */
2799            }
2800            tx_index = tx_index + size;
2801        }
2802        /* Read Data
2803         *  This function will play catch up with respect to the transmit side.
2804         */
2805        /* get status simultaneously */
2806        SPI_Status_Read_FIFO_Status( &rx_empty, &rx_bytes_available );
2807        if ( rx_bytes_available >= size )
2808        { /* read data */
2809            if ( active_rx )
2810            {
2811            for ( index = 0; index < size; index++ ) IO_Read_U8( SPI_DATA, &(((uint8_t *) rx_buffer)[rx_index]) );
2812            }
2813            else
2814            {
2815            for ( index = 0; index < size; index++ ) IO_Read_U8( SPI_DATA, &bit_bucket ); /* toss */
2816            }
2817            rx_index = rx_index + size;
2818        }
2819    }
2820
2821    SPI_Status_Write_FIFO_Status( &tx_full, &tx_empty, &tx_bytes_available );
2822    if ( (false == commit_valid) && (false == tx_empty) )
2823    { /* data has not been transmitted yet */
2824        if ( IDI_CSB_SOFTWARE != csb ) SPI_Commit( 0xFF ); /* does not matter what is written */
2825        /* wait for the buffer to empty */
2826        do
2827        { //TODO: need a timeout and return error code if timeout exceeded.
2828            SPI_Status_Write_FIFO_Status( &tx_full, &tx_empty, &tx_bytes_available );
2829        } while ( false == tx_empty );
2830    }
2831
2832    /* retrieve the remaining read data and don't return until we are done */
2833    while ( rx_index != tx_index )
2834    { //TODO: timeout mechanism????
2835        SPI_Status_Read_FIFO_Status( &rx_empty, &rx_bytes_available );
2836        if ( rx_bytes_available >= size )
2837        { /* read data */
2838            if ( active_rx )
2839            {
2840            for ( index = 0; index < size; index++ ) IO_Read_U8( SPI_DATA, &(((uint8_t *) rx_buffer)[rx_index]) );
2841            }
2842            else
2843            {
2844            for ( index = 0; index < size; index++ ) IO_Read_U8( SPI_DATA, &bit_bucket ); /* toss */
2845            }
2846            rx_index = rx_index + size;
2847        }
2848    }
2849    return SUCCESS;
2850 }
2851 /*****************************************************************************************************//**
2852 @ingroup idi
2853 \brief
2854
2855 This function will write/read virtually any kind of data with almost any kind of chips select wrapping
2856 surrounding the data.
2857
2858 \return a zero if successful, else return an error code.
2859 */
2860 /*****************************************************************************************************//**
2861  * @ingroup idi
2862  * @brief
2863  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2864  *         code is returned.
2865  */
2866 int SPI_Data_Write_Read(    size_t       size,     /**< object size: 1=u8, 2=u16, 4=u16, 8=u64, 16=u128=SPI_FIFO_SIZE */
2867                             size_t       tx_count, /**< object count */
2868                             const void * tx_buffer,
2869                             size_t       rx_size,  /**< object count */
2870                             const void * rx_buffer
2871                         )
2872 {
2873     int                 error_code;
2874     int                 index;
2875     BOOL                active_tx;
2876     BOOL                active_rx;
2877     SPI_CSB_ENUM        csb_copy;
```

```
2878    SPI_CSB_ENUM        csb;
2879    BOOL                csb_buffer_mode_override;
2880
2881 /* TEST FOR VALIDITY OF PARAMETERS */
2882    /* see if there is anything to do */
2883    if ( ( NULL == tx_buffer ) && ( NULL == rx_buffer ) ) return SUCCESS;
2884
2885    if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
2886
2887    /* initialize parameters */
2888    error_code = SPI_Configuration_Chip_Select_Behavior_Get( &csb );
2889    if ( error_code ) return error_code;
2890    //error_code = SPI_Status_Write( &status_tx );
2891    //if ( error_code ) return error_code;
2892    //error_code = SPI_Status_Read( &status_rx );
2893    //if ( error_code ) return error_code;
2894
2895    active_rx = false; /* assume that we toss any data to be read out */
2896    active_tx = false; /* assume that we have no valid data to write  */
2897    if ( NULL != tx_buffer ) active_tx = true;
2898    else                     tx_count  = rx_size;
2899    if ( NULL != rx_buffer ) active_rx = true;
2900    else                     rx_size   = tx_count;
2901
2902
2903    if ( IDI_CSB_uint16_t == csb )
2904    { /* test for even quantity of bytes to transceive */
2905        if ( ( tx_count & 0x01 ) || ( rx_size & 0x01 ) )
2906        { /* odd number of bytes detected for buffers */
2907            return -EC_SPI_BUFFER_SIZE_ODD;
2908        }
2909    }
2910
2911    csb_buffer_mode_override = false;
2912    if ( IDI_CSB_BUFFER == csb )
2913    { /* test for object size */
2914        if ( size > SPI_FIFO_SIZE ) return -EC_SPI_OBJECT_SIZE;
2915    }
2916    else
2917    {
2918        /* test object sizes */
2919        index = SPI_FIFO_SIZE;  /* assumed to be a 2^N number */
2920        while ( index > sizeof( uint16_t ) )
2921        {
2922            if ( size == ( size & index ) )
2923            {
2924                csb_buffer_mode_override = true;
2925                break;
2926            }
2927            index = index >> 1;
2928        }
2929        if ( ( size > 2 ) && ( false == csb_buffer_mode_override ) )
2930        {
2931            return -EC_SPI_OBJECT_SIZE; /* not a power of 2 */
2932        }
2933        else if ( true == csb_buffer_mode_override )
2934        { /* OK, go ahead and change to buffer mode */
2935            csb_copy    = csb;
2936            csb         = IDI_CSB_BUFFER;
2937            error_code = SPI_Configuration_Chip_Select_Behavior_Set( csb );
2938        }
2939    }
2940
2941 /* PERFORM TRANSACTIONS */
2942    switch( csb )
2943    {
2944        case IDI_CSB_SOFTWARE:
2945        case IDI_CSB_uint8_t:
2946        case IDI_CSB_uint16_t:
2947            error_code = SPI_Data_Write_Read_Helper( size,
2948                                                    tx_count,
2949                                                    tx_buffer,
2950                                                    rx_size,
2951                                                    rx_buffer,
2952                                                    active_tx,
2953                                                    active_rx,
2954                                                    csb
2955                                                  );
2956            break;
2957        case IDI_CSB_BUFFER:
2958            for ( index = 0; index < tx_count; index++ )
2959            {
2960                error_code = SPI_Data_Write_Read_Helper( size,
2961                                                        1,
2962                                                        active_tx ? &(((uint8_t *) tx_buffer)[index*size]) : NULL,
2963                                                        1,
2964                                                        active_rx ? &(((uint8_t *) rx_buffer)[index*size]) : NULL,
2965                                                        active_tx,
2966                                                        active_rx,
2967                                                        csb
2968                                                      );
2969            }
2970            if ( csb_buffer_mode_override )
```

```
2971            { /* restore to original csb */
2972                error_code = SPI_Configuration_Chip_Select_Behavior_Set( csb_copy );
2973            }
2974            break;
2975        default:
2976            return -EC_SPI_CSB_OUT_OF_RANGE;
2977            //break;
2978    }
2979    return SUCCESS;
2980}
2981/*****************************************************************************************************//**
2982@ingroup idi
2983\brief
2984
2985Special case of Write/Read that has a function signature same as fread() or fwrite().
2986
2987\param[in] cfg pass in the configuration to be written to hardware.
2988\return a nonzero if successful, else return zero.
2989*/
2990/*****************************************************************************************************//**
2991 * @ingroup idi
2992 * @brief
2993 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
2994 *         code is returned.
2995 */
2996
2997int SPI_Data_Write( const void *    tx_buffer,
2998                    size_t          size,       /**< object size  */
2999                    size_t          tx_count,   /**< object count */
3000                    FILE *          fd_log      /**< set to NULL if no file logging */
3001                  )
3002{
3003    int error_code;
3004    error_code = SPI_Data_Write_Read(   size,
3005                                        tx_count,
3006                                        tx_buffer,
3007                                        0,          /* nothing to receive */
3008                                        NULL        /* nothing to receive */
3009                                    );
3010    if ( error_code ) return error_code;
3011
3012    if ( NULL != fd_log )
3013    {
3014        error_code = fwrite( tx_buffer, size, tx_count, fd_log );
3015    }
3016    return error_code;
3017}
3018/*****************************************************************************************************//**
3019@ingroup idi
3020\brief
3021
3022Special case of Write/Read that has a function signature same as fread() or fwrite().
3023
3024\param[in] cfg pass in the configuration to be written to hardware.
3025\return a nonzero if successful, else return zero.
3026*/
3027/*****************************************************************************************************//**
3028 * @ingroup idi
3029 * @brief
3030 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3031 *         code is returned.
3032 */
3033int SPI_Data_Read(  const void *    rx_buffer,
3034                    size_t          size,       /**< object size  */
3035                    size_t          rx_size,    /**< object count */
3036                    FILE *          fd_log      /**< set to NULL if no file logging */
3037                  )
3038{
3039    int error_code;
3040    error_code = SPI_Data_Write_Read(   size,
3041                                        0,      /* nothing to transmit */
3042                                        NULL,   /* nothing to transmit */
3043                                        rx_size,
3044                                        rx_buffer
3045                                    );
3046    if ( error_code ) return error_code;
3047
3048    if ( NULL != fd_log )
3049    {
3050        error_code = fwrite( rx_buffer, size, rx_size, fd_log );
3051    }
3052    return error_code;
3053}
3054
3055
3056
3057
3058/*****************************************************************************************************/
3059/*****************************************************************************************************/
3060/*****************************************************************************************************/
3061/*    < < < <  F R A M   F U N C T I O N S  > > > >                                                  */
3062
3063
```

```
3064 /*****************************************************************************************************//**
3065  * @ingroup idi
3066  * @brief
3067  * FRAM Density current in use.
3068  */
3069 enum { FRAM_DENSITY_BYTES = 8192 };
3070
3071 /*****************************************************************************************************//**
3072  * @ingroup idi
3073  * @brief
3074  * FRAM Write Enable Latch Set command (WREN)
3075  *
3076  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3077  *         code is returned.
3078  */
3079 int FRAM__Write_Enable_Latch_Set( void )
3080 {
3081     uint8_t tx_buf[1] = { 0x06 }; /* opcode: WREN = 0x06 */
3082     SPI_Configuration_Chip_Select_Behavior_Set( IDI_CSB_BUFFER );
3083     return SPI_Data_Write_Read( sizeof( uint8_t ), 1, tx_buf, 0, NULL );
3084 }
3085 /*****************************************************************************************************//**
3086  * @ingroup idi
3087  * @brief
3088  * FRAM Write Latch disable (or clear) command (WRDI).
3089  *
3090  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3091  *         code is returned.
3092  */
3093 int FRAM__Write_Disable( void )
3094 {
3095     uint8_t tx_buf[1] = { 0x04 }; /* opcode: WRDI = 0x04 */
3096     SPI_Configuration_Chip_Select_Behavior_Set( IDI_CSB_BUFFER );
3097     return SPI_Data_Write_Read( sizeof( uint8_t ), 1, tx_buf, 0, NULL );
3098 }
3099 /*****************************************************************************************************//**
3100  * @ingroup idi
3101  * @brief
3102  * Read the FRAM status register and output the value.
3103  *
3104  * @param status A pointer to the destination location of the status data.
3105  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3106  *         code is returned.
3107  */
3108 int FRAM__Read_Status_Register( uint8_t * status )
3109 {
3110     int     error_code;
3111     uint8_t tx_buf[2] = { 0x05, 0x00 }; /* opcode: RDSR = 0x04 */
3112     uint8_t rx_buf[2];
3113
3114     SPI_Configuration_Chip_Select_Behavior_Set( IDI_CSB_BUFFER );
3115     error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 2, tx_buf, 2, rx_buf );
3116     if ( error_code ) return error_code;
3117     *status = rx_buf[1];
3118     return SUCCESS;
3119 }
3120 /*****************************************************************************************************//**
3121  * @ingroup idi
3122  * @brief
3123  * Write to the FRAM status register.
3124  *
3125  * @param[in]  FRAM status value to be written.
3126  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3127  *         code is returned.
3128  */
3129 int FRAM__Write_Status_Register( uint8_t status )
3130 {
3131     int     error_code;
3132     uint8_t tx_buf[2] = { 0x05, 0x00 }; /* opcode: RDSR = 0x04 */
3133     uint8_t rx_buf[2];
3134
3135     tx_buf[1] = status;
3136     SPI_Configuration_Chip_Select_Behavior_Set( IDI_CSB_BUFFER );
3137     error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 2, tx_buf, 2, rx_buf );
3138     if ( error_code ) return error_code;
3139     return SUCCESS;
3140 }
3141 /*****************************************************************************************************//**
3142  * @ingroup idi
3143  * @brief
3144  * Reads data from FRAM memory to the output buffer.
3145  *
3146  * @param[in] address   Starting FRAM address
3147  * @param[in] count     Number of bytes to transfer
3148  * @param[out] buffer   Destination buffer in which to store the data
3149  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3150  *         code is returned.
3151  */
3152 int FRAM__Memory_Read( uint16_t address, size_t count, uint8_t * buffer )
3153 {
3154     int             error_code;
3155 //  SPI_CSB_ENUM    csb_copy;
3156     uint8_t         tx_buf[3] = { 0x03, 0x00, 0x00 }; /* opcode: READ = 0x03 */
```

```
3157    //uint8_t rx_buf[3];
3158
3159    tx_buf[1] = (uint8_t)( address & 0xFF );
3160    tx_buf[2] = (uint8_t)( address >>   8 );
3161
3162//  /* retain an existing copy of the actual csb value */
3163//  error_code = SPI_Configuration_Chip_Select_Behavior_Get( &csb_copy );
3164//  if ( error_code ) return error_code;
3165    /* over-ride and set it to what we wish it to be */
3166    error_code = SPI_Configuration_Chip_Select_Behavior_Set( IDI_CSB_SOFTWARE );
3167    if ( error_code ) return error_code;
3168
3169    SPI_Commit( 1 );
3170
3171    error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 3, tx_buf, 0, NULL );
3172    if ( error_code ) return error_code;
3173
3174    error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 0, NULL, count, buffer );
3175    if ( error_code ) return error_code;
3176
3177    while ( SPI_Status_Write_FIFO_Is_Not_Empty() ); /* wait for buffer to empty */
3178    SPI_Commit( 0 );
3179
3180//  /* restore the csb */
3181//  error_code = SPI_Configuration_Chip_Select_Behavior_Set( csb_copy );
3182//  if ( error_code ) return error_code;
3183
3184    return SUCCESS;
3185}
3186/*****************************************************************************************************//**
3187 * @ingroup idi
3188 * @brief
3189 * Writes data from buffer to FRAM memory.
3190 *
3191 * @param[in] address   Starting FRAM address
3192 * @param[in] count     Number of bytes to transfer
3193 * @param[out] buffer   Source buffer from which data will be transfered to FRAM
3194 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3195 *         code is returned.
3196 */
3197int FRAM__Memory_Write( uint16_t address, size_t count, uint8_t * buffer )
3198{
3199    int             error_code;
3200//  SPI_CSB_ENUM    csb_copy;
3201    uint8_t         tx_buf[3] = { 0x02, 0x00, 0x00 }; /* opcode: WRITE = 0x02 */
3202
3203    tx_buf[1] = (uint8_t)( address & 0xFF );
3204    tx_buf[2] = (uint8_t)( address >>   8 );
3205
3206//  /* retain an existing copy of the actual csb value */
3207//  error_code = SPI_Configuration_Chip_Select_Behavior_Get( &csb_copy );
3208//  if ( error_code ) return error_code;
3209    /* over-ride and set it to what we wish it to be */
3210    error_code = SPI_Configuration_Chip_Select_Behavior_Set( IDI_CSB_SOFTWARE );
3211    if ( error_code ) return error_code;
3212
3213    SPI_Commit( 1 );
3214
3215    error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 3, tx_buf, 0, NULL );
3216    if ( error_code ) return error_code;
3217
3218    error_code = SPI_Data_Write_Read( sizeof( uint8_t ), count, buffer, 0, NULL );
3219    if ( error_code ) return error_code;
3220
3221    while ( SPI_Status_Write_FIFO_Is_Not_Empty() ); /* wait for buffer to empty */
3222    SPI_Commit( 0 );
3223
3224//  /* restore the csb */
3225//  error_code = SPI_Configuration_Chip_Select_Behavior_Set( csb_copy );
3226//  if ( error_code ) return error_code;
3227    return SUCCESS;
3228}
3229/*****************************************************************************************************//**
3230 * @ingroup idi
3231 * @brief
3232 *
3233 * @param[out] id  The 32-bit ID register read from the FRAM.  This appears to be only available
3234 *                 with Fujistu parts.
3235 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3236 *         code is returned.
3237 */
3238int FRAM__Read_ID( uint32_t * id )
3239{
3240    int         error_code;
3241    uint8_t     tx_buf[5] = { 0x9F, 0x00, 0x00, 0x00, 0x00 }; /* opcode: RDID = 0x9F */
3242    uint8_t     rx_buf[5];
3243
3244    SPI_Configuration_Chip_Select_Behavior_Set( IDI_CSB_BUFFER );
3245    error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 5, tx_buf, 5, rx_buf );
3246    if ( error_code ) return error_code;
3247
3248    {
3249        int id_scratch = 0;
```

```
3250        int index;
3251
3252        for ( index = 4; index > 0; index-- )
3253        { /* assuming MSB first */
3254            id_scratch = ( id_scratch << 8 ) | ( (uint32_t) rx_buf[index] );
3255        }
3256        *id = id_scratch;
3257    }
3258
3259    return SUCCESS;
3260}
3261/*****************************************************************************************************//**
3262@ingroup idi
3263\brief
3264
3265This function will be used when creating a memory pool so that as blocks are
3266allocated one can determine if we have an issue outside of any allocated space
3267(i.e. overflows and so on).
3268
3269\param[in] cfg pass in the configuration to be written to hardware.
3270\return a nonzero if successful, else return zero.
3271*/
3272/*****************************************************************************************************//**
3273 * @ingroup idi
3274 * @brief
3275 * Writes a repeating pattern to the entire FRAM memory array.  the pattern is obtained
3276 * from the buffer.  If the buffer is NULL, then all zeros are written to the FRAM.
3277 *
3278 * @param[in] count  Length in bytes of the pattern found within the buffer
3279 * @param[in] buffer input buffer containing the repeat pattern to be written to FRAM
3280 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3281 *         code is returned.
3282 */
3283int FRAM_Set( size_t count, uint8_t * buffer )
3284{
3285    int         error_code;
3286    int         block_count;
3287    int         block_remainder;
3288    uint16_t    address;
3289    uint32_t    id;
3290
3291    error_code = FRAM__Read_ID( &id );
3292    if ( error_code ) return error_code;
3293
3294    address = 0;
3295
3296    if ( count > 1 )
3297    { /* */
3298        block_count     = ((int) FRAM_DENSITY_BYTES) / ((int) count );
3299        block_remainder = ((int) FRAM_DENSITY_BYTES) - ( block_count * ((int) FRAM_BLOCK_SIZE) );
3300    }
3301    else
3302    { /* only one fill character, so we create a buffer of it to make things a bit faster */
3303        int         index;
3304        const int   block_size = FRAM_BLOCK_SIZE;
3305        /* prefill */
3306        if ( NULL == buffer )
3307        {
3308            for ( index = 0; index < block_size; index++ ) idi_dataset.fram_block[index]= 0;
3309        }
3310        else
3311        {
3312            for ( index = 0; index < block_size; index++ ) idi_dataset.fram_block[index]= buffer[0];
3313        }
3314        block_count     = ((int) FRAM_DENSITY_BYTES) / block_size;
3315        block_remainder = ((int) FRAM_DENSITY_BYTES) - ( block_count * block_size );
3316        buffer          = idi_dataset.fram_block;
3317    }
3318
3319    while ( block_count > 0 )
3320    {
3321        error_code = FRAM__Memory_Write( address, ((int) count), buffer );
3322        address += (uint16_t) count;
3323        block_count--;
3324    }
3325
3326    if ( block_remainder > 0 )
3327    {
3328        error_code = FRAM__Memory_Write( address, block_remainder, buffer );
3329        if ( error_code ) return error_code;
3330    }
3331    return SUCCESS;
3332}
3333/*****************************************************************************************************//**
3334 * @ingroup idi
3335 * @brief
3336 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3337 *         code is returned.
3338 */
3339int FRAM_Report( uint16_t address, size_t length, FILE * out )
3340{
3341    int         error_code;
3342    const int   block_size = HEX_DUMP_BYTES_PER_LINE;
```

```
3343    size_t      block_count;
3344    size_t      block_remainder;
3345
3346    block_count = ((size_t) length) / block_size;
3347    block_remainder = ((size_t) length) - ( block_count * block_size );
3348
3349    while ( block_count > 0 )
3350    { /* output a line at a time */
3351        error_code = FRAM__Memory_Read( address, block_size, idi_dataset.fram_block );
3352        error_code = Hex_Dump_Line( address, block_size, idi_dataset.fram_block, out );
3353        address += block_size;
3354        block_count--;
3355    }
3356
3357    if ( block_remainder > 0 )
3358    { /* output any remaining portion */
3359        error_code = FRAM__Memory_Read( address, block_remainder, idi_dataset.fram_block );
3360        error_code = Hex_Dump_Line( address, block_remainder, idi_dataset.fram_block, out );
3361        if ( error_code ) return error_code;
3362    }
3363
3364    return SUCCESS;
3365 }
3366 /*****************************************************************************************************//**
3367  * @ingroup idi
3368  * @brief
3369  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3370  *         code is returned.
3371  */
3372 int FRAM_Memory_To_File( uint16_t address, size_t length, FILE * binary )
3373 {
3374     int         error_code;
3375     size_t      block_count;
3376     size_t      block_remainder;
3377
3378     block_count = length / ((size_t) FRAM_BLOCK_SIZE);
3379     block_remainder = length - ( block_count * ((size_t) FRAM_BLOCK_SIZE) );
3380
3381     while ( block_count > 0 )
3382     {
3383         error_code = FRAM__Memory_Read( address, ((size_t) FRAM_BLOCK_SIZE), idi_dataset.fram_block );
3384         fwrite( idi_dataset.fram_block, 1, ((size_t) FRAM_BLOCK_SIZE), binary );
3385         address += FRAM_BLOCK_SIZE;
3386         block_count--;
3387     }
3388
3389     if ( block_remainder > 0 )
3390     {
3391         error_code = FRAM__Memory_Read( address, block_remainder, idi_dataset.fram_block );
3392         fwrite( idi_dataset.fram_block, 1, block_remainder, binary );
3393         if ( error_code ) return error_code;
3394     }
3395     return SUCCESS;
3396 }
3397 /*****************************************************************************************************//**
3398  * @ingroup idi
3399  * @brief
3400  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3401  *         code is returned.
3402  */
3403 int FRAM_File_To_Memory( uint16_t address, size_t length, FILE * binary )
3404 {
3405     int     error_code;
3406     size_t  count_read;
3407     size_t  count_total;
3408     size_t  count_actual;
3409
3410     count_total = 0;
3411     count_read  = FRAM_BLOCK_SIZE;
3412     if ( 0 == length )
3413     {
3414         do
3415         {
3416             count_actual = fread( idi_dataset.fram_block, 1, count_read, binary );
3417             error_code   = FRAM__Memory_Write( address, count_read, idi_dataset.fram_block );
3418             if ( error_code ) return error_code;
3419             count_total += count_actual;
3420             if ( count_actual != count_read ) count_read = 0; /* must be at end of file */
3421         } while ( count_read > 0 );
3422     }
3423     else
3424     {
3425         if ( length < count_read ) count_read = length;
3426         do
3427         {
3428             count_actual = fread( idi_dataset.fram_block, 1, count_read, binary );
3429             error_code   = FRAM__Memory_Write( address, count_read, idi_dataset.fram_block );
3430             if ( error_code ) return error_code;
3431             count_total += count_actual;
3432             length      -= count_actual;
3433             if ( count_actual != count_read ) count_read = 0; /* must be at end of file */
3434             if ( length < count_read ) count_read = length;
3435         } while ( count_read > 0 );
```

```
3436    }
3437    return SUCCESS;
3438 }
3439
3440
3441 /*****************************************************************************************************/
3442 /*****************************************************************************************************/
3443 /*****************************************************************************************************/
3444
3445
3446 /*****************************************************************************************************//**
3447  * @ingroup idi
3448  * @brief
3449  * Data structure used to decode command line operation.
3450  */
3451 struct command_line
3452 {
3453     struct command_line * link;                     /* link to next lower level data structure        */
3454     int ( * cmd_fnc )( int argc, char * argv[] );   /* function to call to process arguments further  */
3455     char *  name;                                   /* name of argument/command word                  */
3456     char *  help;                                   /* very brief help string associated with command */
3457 };
3458 /*****************************************************************************************************/
3459
3460
3461 /*****************************************************************************************************/
3462 /*****************************************************************************************************/
3463 /*****************************************************************************************************/
3464 /*    < < < <  S P I  C O M M A N D  F U N C T I O N S  > > > >                                       */
3465
3466
3467 /*****************************************************************************************************//**
3468  * @ingroup idi
3469  * @brief
3470  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3471  *         code is returned.
3472  */
3473 static int IDI_CMD__SPI_ID( int argc, char * argv[] )
3474 { /*  idi spi id */
3475     uint16_t    id;
3476     (void)      argc;
3477     (void)      argv;
3478
3479     SPI_ID_Get( &id );
3480     printf( "SPI ID: 0x%04X\n", id );
3481     return SUCCESS;
3482 }
3483 /*****************************************************************************************************//**
3484  * @ingroup idi
3485  * @brief
3486  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3487  *         code is returned.
3488  */
3489 static int IDI_CMD__SPI_Config_Get( int argc, char * argv[] )
3490 { /*  idi spi cfg */
3491     int     error_code;
3492     struct  spi_cfg cfg;
3493     (void)  argc;
3494     (void)  argv;
3495
3496     error_code = SPI_Configuration_Get( &cfg );
3497     if ( error_code ) return error_code;
3498
3499     error_code = SPI_Report_Configuration_Text( &cfg, stdout );
3500     if ( error_code ) return error_code;
3501
3502     printf( "\n" );
3503     return SUCCESS;
3504 }
3505 /*****************************************************************************************************//**
3506  * @ingroup idi
3507  * @brief
3508  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3509  *         code is returned.
3510  */
3511 static int IDI_CMD__SPI_Config_Clock_Hz( int argc, char * argv[] )
3512 { /*  idi spi clk [<freq_hz>] */
3513     int      error_code;
3514     double   clock_request_hz;
3515     double   clock_actual_hz;
3516     double   error;
3517     uint16_t hci;
3518     struct   spi_cfg cfg;
3519
3520     /* pull current configuration from the hardware -- allows for warm restore so to speak */
3521     error_code = SPI_Configuration_Get( &cfg );
3522     if ( error_code ) return error_code;
3523
3524     if ( argc < 1 )
3525     { /* read */
3526         printf( "SPI CLK: %f hz\n", cfg.clock_hz );
3527     }
3528     else
```

```
3529    { /* write */
3530        clock_request_hz = atof( argv[0] );
3531        error_code = SPI_Calculate_Clock( clock_request_hz, &clock_actual_hz, &error, &hci );
3532        if ( error_code ) return error_code;
3533        //cfg.half_clock_interval = hci;
3534        /* commit configuration to hardware */
3535        cfg.clock_hz = clock_actual_hz;
3536        error_code   = SPI_Configuration_Set( &cfg );
3537        if ( error_code ) return error_code;
3538        printf( "OK\n" );
3539    }
3540    return SUCCESS;
3541 }
3542 /*****************************************************************************************************//**
3543  * @ingroup idi
3544  * @brief
3545  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3546  *         code is returned.
3547  */
3548 static int IDI_CMD__SPI_Config_End_Cycle_Delay_Sec( int argc, char * argv[] )
3549 { /*  idi spi ecd [<time sec>] */
3550     int      error_code;
3551     double   request_sec;
3552     double   actual_sec;
3553     double   error;
3554     uint8_t  ecd;
3555     struct   spi_cfg cfg;
3556
3557     /* pull current configuration from the hardware -- allows for warm restore so to speak */
3558     error_code = SPI_Configuration_Get( &cfg );
3559     if ( error_code ) return error_code;
3560
3561     if ( argc < 1 )
3562     { /* read */
3563         printf( "SPI ECD: %g sec\n", ( cfg.end_delay_ns * 1.0e-9 ) );
3564     }
3565     else
3566     { /* write */
3567         request_sec = atof( argv[0] );
3568         error_code = SPI_Calculate_End_Cycle_Delay( SPI_Calculate_Half_Clock_Interval_Sec( cfg.half_clock_interval ),
3569                                                     request_sec,
3570                                                     &actual_sec,
3571                                                     &error,
3572                                                     &ecd
3573                                                   );
3574         if ( error_code ) return error_code;
3575         cfg.end_delay_ns = actual_sec * 1.0e9;
3576         /* commit configuration to hardware */
3577         error_code = SPI_Configuration_Set( &cfg );
3578         if ( error_code ) return error_code;
3579         printf( "OK\n" );
3580     }
3581     return SUCCESS;
3582 }
3583 /*****************************************************************************************************//**
3584 @ingroup idi
3585 \brief
3586         CPOL    CPHA    MODE
3587          0       0       0
3588          1       0       1
3589          2       1       0
3590          3       1       1
3591 */
3592 /*****************************************************************************************************//**
3593  * @ingroup idi
3594  * @brief
3595  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3596  *         code is returned.
3597  */
3598 static int IDI_CMD__SPI_Config_Mode( int argc, char * argv[] )
3599 { /*  idi spi mode [0/1/2/3] */
3600     int      error_code;
3601     int      mode;
3602     struct   spi_cfg cfg;
3603
3604     /* pull current configuration from the hardware -- allows for warm restore so to speak */
3605     error_code = SPI_Configuration_Get( &cfg );
3606     if ( error_code ) return error_code;
3607
3608     if ( argc < 1 )
3609     { /* read */
3610         if      ( (false == cfg.sclk_polarity ) && (false == cfg.sclk_phase) ) mode = 0;
3611         else if ( (false == cfg.sclk_polarity ) && (true  == cfg.sclk_phase) ) mode = 1;
3612         else if ( (true  == cfg.sclk_polarity ) && (false == cfg.sclk_phase) ) mode = 2;
3613         else if ( (true  == cfg.sclk_polarity ) && (true  == cfg.sclk_phase) ) mode = 3;
3614         printf( "SPI MODE: %d\n", mode );
3615     }
3616     else
3617     { /* write */
3618         mode = (int) strtol( argv[0], NULL, 0 );
3619         switch ( mode )
3620         {
3621             case 0: cfg.sclk_polarity = false;  cfg.sclk_phase = false;     break;
```

```
3622            case 1: cfg.sclk_polarity = false;  cfg.sclk_phase = true;      break;
3623            case 2: cfg.sclk_polarity = true;   cfg.sclk_phase = false;     break;
3624            case 3: cfg.sclk_polarity = true;   cfg.sclk_phase = true;      break;
3625        }
3626        /* commit configuration to hardware */
3627        error_code = SPI_Configuration_Set( &cfg );
3628        if ( error_code ) return error_code;
3629        printf( "OK\n" );
3630    }
3631    return SUCCESS;
3632}
3633/*****************************************************************************************************//**
3634 * @ingroup idi
3635 * @brief
3636 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3637 *         code is returned.
3638 */
3639static int IDI_CMD__SPI_Config_SDI_Polarity( int argc, char * argv[] )
3640{ /*  idi spi sdi [<true/1/false/0>] */
3641    int      error_code;
3642//  BOOL     value;
3643    struct   spi_cfg cfg;
3644
3645    /* pull current configuration from the hardware -- allows for warm restore so to speak */
3646    error_code = SPI_Configuration_Get( &cfg );
3647    if ( error_code ) return error_code;
3648
3649    if ( argc < 1 )
3650    { /* read */
3651        printf( "SPI SDI POLARITY: %s\n", cfg.sdi_polarity ? "true" : "false" );
3652    }
3653    else
3654    { /* write */
3655        cfg.sdi_polarity = String_To_Bool( argv[0] );
3656        /* commit configuration to hardware */
3657        error_code = SPI_Configuration_Set( &cfg );
3658        if ( error_code ) return error_code;
3659        printf( "OK\n" );
3660    }
3661    return SUCCESS;
3662}
3663/*****************************************************************************************************//**
3664 * @ingroup idi
3665 * @brief
3666 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3667 *         code is returned.
3668 */
3669static int IDI_CMD__SPI_Config_SDO_Polarity( int argc, char * argv[] )
3670{ /*  idi spi sdo [<true/1/false/0>] */
3671    int      error_code;
3672//  BOOL     value;
3673    struct   spi_cfg cfg;
3674
3675    /* pull current configuration from the hardware -- allows for warm restore so to speak */
3676    error_code = SPI_Configuration_Get( &cfg );
3677    if ( error_code ) return error_code;
3678
3679    if ( argc < 1 )
3680    { /* read */
3681        printf( "SPI SDO POLARITY: %s\n", cfg.sdo_polarity ? "true" : "false" );
3682    }
3683    else
3684    { /* write */
3685        cfg.sdo_polarity = String_To_Bool( argv[0] );
3686        /* commit configuration to hardware */
3687        error_code = SPI_Configuration_Set( &cfg );
3688        if ( error_code ) return error_code;
3689        printf( "OK\n" );
3690    }
3691    return SUCCESS;
3692}
3693/*****************************************************************************************************//**
3694 * @ingroup idi
3695 * @brief
3696 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3697 *         code is returned.
3698 */
3699static int IDI_CMD__SPI_Config_SDIO_Wrap( int argc, char * argv[] )
3700{ /*  idi spi wrap [<true/1/false/0>] */
3701    int      error_code;
3702//  BOOL     value;
3703    struct   spi_cfg cfg;
3704
3705    /* pull current configuration from the hardware -- allows for warm restore so to speak */
3706    error_code = SPI_Configuration_Get( &cfg );
3707    if ( error_code ) return error_code;
3708
3709    if ( argc < 1 )
3710    { /* read */
3711        printf( "SPI wrap: %s\n", cfg.sdio_wrap ? "true" : "false" );
3712    }
3713    else
3714    { /* write */
```

```
3715        cfg.sdio_wrap = String_To_Bool( argv[0] );
3716        /* commit configuration to hardware */
3717        error_code = SPI_Configuration_Set( &cfg );
3718        if ( error_code ) return error_code;
3719        printf( "OK\n" );
3720    }
3721    return SUCCESS;
3722}
3723/**************************************************************************************************//**
3724 * @ingroup idi
3725 * @brief
3726 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3727 *         code is returned.
3728 */
3729static int IDI_CMD__SPI_Config_Chip_Select_Behavior( int argc, char * argv[] )
3730{ /*  idi spi mode [0/1/2/3/software/buffer/uint8_t/uint16_t] */
3731    int      error_code;
3732//  int      csb;
3733    struct   spi_cfg cfg;
3734
3735    /* pull current configuration from the hardware -- allows for warm restore so to speak */
3736    error_code = SPI_Configuration_Get( &cfg );
3737    if ( error_code ) return error_code;
3738
3739    if ( argc < 1 )
3740    { /* read */
3741        printf( "SPI CSB: ");
3742        switch ( cfg.chip_select_behavior )
3743        {
3744            case IDI_CSB_SOFTWARE:  printf( "software" );   break;
3745            case IDI_CSB_BUFFER:    printf( "buffer"   );   break;
3746            case IDI_CSB_uint8_t:       printf( "uint8_t"    ); break;
3747            case IDI_CSB_uint16_t:  printf( "uint16_t"   ); break;
3748            default:                printf( "undefined");   break;
3749        }
3750        printf( "\n" );
3751    }
3752    else
3753    { /* write */
3754        if      ( 0 == strcmpi( "software", argv[0] ) ) cfg.chip_select_behavior = 0;
3755        else if ( 0 == strcmpi( "buffer",   argv[0] ) ) cfg.chip_select_behavior = 1;
3756        else if ( 0 == strcmpi( "uint8_t",    argv[0] ) ) cfg.chip_select_behavior = 2;
3757        else if ( 0 == strcmpi( "uint16_t",   argv[0] ) ) cfg.chip_select_behavior = 3;
3758        else
3759        {
3760            cfg.chip_select_behavior = (SPI_CSB_ENUM) strtol( argv[0], NULL, 0 );
3761            switch ( cfg.chip_select_behavior )
3762            {
3763                case IDI_CSB_SOFTWARE:
3764                case IDI_CSB_BUFFER:
3765                case IDI_CSB_uint8_t:
3766                case IDI_CSB_uint16_t:
3767                    break;
3768                default:
3769                    error_code = -EC_SPI_CSB_OUT_OF_RANGE;
3770                    break;
3771            }
3772        }
3773         if ( error_code ) return error_code;
3774        /* commit configuration to hardware */
3775        error_code = SPI_Configuration_Set( &cfg );
3776        if ( error_code ) return error_code;
3777        printf( "OK\n" );
3778    }
3779    return SUCCESS;
3780}
3781/**************************************************************************************************//**
3782 * @ingroup idi
3783 * @brief
3784 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3785 *         code is returned.
3786 */
3787static int IDI_CMD__SPI_Status( int argc, char * argv[] )
3788{ /*  idi spi status [rx] [tx] ...  */
3789    int                 error_code;
3790    int                 index;
3791    struct spi_status   status;
3792
3793    if ( argc < 1 )
3794    {
3795        error_code = SPI_Status_Read( &status );
3796        if ( error_code ) return error_code;
3797        SPI_Report_Status_Text( &status, stdout );
3798    }
3799    else
3800    {
3801        for ( index = 0; index < argc; index++ )
3802        {
3803            if ( 0 == strcmpi( "rx", argv[index] ) )
3804            {
3805                error_code = SPI_Status_Read( &status );
3806                if ( error_code ) return error_code;
3807                SPI_Report_Status_Text( &status, stdout );
```

```
3808            }
3809            else if ( 0 == strcmpi( "tx", argv[index] ) )
3810            {
3811                error_code = SPI_Status_Write( &status );
3812                if ( error_code ) return error_code;
3813                SPI_Report_Status_Text( &status, stdout );
3814            }
3815        }
3816    }
3817    return SUCCESS;
3818}
3819/**************************************************************************************************//**
3820 * @ingroup idi
3821 * @brief
3822 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3823 *         code is returned.
3824 */
3825static int IDI_CMD__SPI_Data( int argc, char * argv[] )
3826{ /*  idi spi data [byte] [character] ... */
3827    int                 error_code;
3828    size_t              index;
3829    size_t              count;
3830    size_t              transfer_count; /* */
3831    size_t              lines;
3832    uint8_t *           bp;             /* buffer pointer */
3833    uint8_t             tx_buffer[SPI_FIFO_SIZE];
3834    uint8_t             rx_buffer[SPI_FIFO_SIZE];
3835
3836
3837    if ( argc < 1 ) return -EC_PARAMETER;
3838
3839    index = 1;
3840    transfer_count = argc - 1;
3841    if ( transfer_count > SPI_FIFO_SIZE )
3842    {
3843        transfer_count = SPI_FIFO_SIZE;
3844        printf( "Warning: ignored %d values\n", argc - 1 - SPI_FIFO_SIZE );
3845    }
3846    count = transfer_count;
3847    while ( count != 0 )
3848    {
3849        tx_buffer[index-1] = (uint8_t) strtol( argv[index], NULL, 0 );
3850        count--; index++;
3851    }
3852    error_code = SPI_Data_Write_Read( sizeof( uint8_t ), transfer_count, tx_buffer, transfer_count, rx_buffer );
3853    if ( error_code ) return error_code;
3854
3855    lines = transfer_count / HEX_DUMP_BYTES_PER_LINE;
3856    if ( 0 == ( transfer_count - lines * HEX_DUMP_BYTES_PER_LINE ) ) lines = lines - 1;
3857    for ( index = 0; index <= lines; index++ )
3858    {
3859        bp = &(rx_buffer[index * HEX_DUMP_BYTES_PER_LINE]);
3860        if ( transfer_count < HEX_DUMP_BYTES_PER_LINE )
3861        {
3862            Hex_Dump_Line( 0, transfer_count, bp, stdout );
3863        }
3864        else
3865        {
3866            Hex_Dump_Line( 0, HEX_DUMP_BYTES_PER_LINE, bp, stdout );
3867        }
3868    }
3869    return SUCCESS;
3870}
3871/**************************************************************************************************//**
3872 * @ingroup idi
3873 * @brief
3874 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3875 *         code is returned.
3876 */
3877static int IDI_CMD__SPI_FIFO( int argc, char * argv[] )
3878{ /*  idi spi data [byte] [character] ... */
3879    int                 error_code;
3880    int                 index;
3881    int                 count;
3882    int                 read_count; /* */
3883    uint8_t             data_temp;
3884    uint8_t             tx_buffer[SPI_FIFO_SIZE];
3885    uint8_t             rx_buffer[SPI_FIFO_SIZE];
3886
3887
3888    read_count = 0;
3889    if ( argc < 1 )
3890    {
3891        read_count = SPI_FIFO_SIZE;
3892    }
3893    else
3894    {
3895        if      ( 0 == strcmpi( "rx", argv[0] ) )
3896        {
3897            if ( argc > 1)
3898            {
3899                if ( 0 == strcmpi( "all", argv[1] ) ) read_count = SPI_FIFO_SIZE;
3900                else read_count = (int) strtol( argv[1], NULL, 0 );
```

```
3901            }
3902        }
3903        else if ( 0 == strcmpi( "tx", argv[0] ) )
3904        {
3905            char * endptr;
3906
3907            index = 1;
3908            count = argc - 1;
3909            if ( count > SPI_FIFO_SIZE )
3910            {
3911                count = SPI_FIFO_SIZE;
3912                printf( "Warning: ignored %d values\n", argc - 1 - SPI_FIFO_SIZE );
3913            }
3914            while ( count > 0 )
3915            {
3916                data_temp = (uint8_t) strtol( argv[index], &endptr, 0 );
3917                if ( endptr == argv[index] ) data_temp = (uint8_t) argv[index][0];
3918                tx_buffer[index-1] = data_temp;
3919                count--; index++;
3920            }
3921            error_code = SPI_FIFO_Write( (void *) tx_buffer, sizeof( uint8_t ), count, NULL );
3922            if ( error_code ) return error_code;
3923
3924            printf( "OK\n" );
3925            return error_code;
3926        }
3927        else if ( 0 == strcmpi( "commit", argv[0] ) )
3928        {
3929            if ( argc > 2 )
3930            {
3931                if ( String_To_Bool( argv[1] ) ) SPI_Commit( 0xFF );
3932                else                             SPI_Commit( 0x00 );
3933
3934                printf( "OK\n" );
3935                error_code = SUCCESS;
3936            }
3937            else
3938            {
3939//TODO: add ability to read-back the chip select under certain conditions.
3940                error_code = -EC_PARAMETER_MISSING;
3941            }
3942            return error_code;
3943        }
3944    }
3945
3946    /* wait for transmit data to empty out */
3947    while ( SPI_Status_Write_FIFO_Is_Not_Empty() ) { /* do nothing */ }
3948
3949    if ( read_count > 0 )
3950    {
3951        int       lines;
3952        uint8_t * bp; /* buffer pointer */
3953
3954        if ( SPI_Status_Read_FIFO_Is_Not_Empty() )
3955        {
3956            error_code = SPI_FIFO_Read( (void *) rx_buffer, sizeof( uint8_t ), read_count, NULL );
3957            if ( error_code ) return error_code;
3958
3959            lines = read_count / HEX_DUMP_BYTES_PER_LINE;
3960            if ( 0 == ( read_count - lines * HEX_DUMP_BYTES_PER_LINE ) ) lines = lines - 1;
3961            for ( index = 0; index <= lines; index++ )
3962            {
3963                bp = &(rx_buffer[index * HEX_DUMP_BYTES_PER_LINE]);
3964                if ( read_count < HEX_DUMP_BYTES_PER_LINE )
3965                {
3966                    Hex_Dump_Line( 0, read_count, bp, stdout );
3967                }
3968                else
3969                {
3970                    Hex_Dump_Line( 0, HEX_DUMP_BYTES_PER_LINE, bp, stdout );
3971                }
3972            }
3973        }
3974        else
3975        {
3976            printf( "FIFO Empty\n" );
3977        }
3978    }
3979    return SUCCESS;
3980}
3981/*****************************************************************************************************//**
3982 * @ingroup idi
3983 * @brief
3984 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
3985 *         code is returned.
3986 */
3987static int IDI_CMD__SPI_Commit( int argc, char * argv[] )
3988{ /*  idi spi commit [<true/1/false/0>] */
3989    int      error_code;
3990    uint8_t  chip_select;
3991
3992    if      ( argc < 1                  ) chip_select = 0x01;
3993    else if ( String_To_Bool( argv[0] ) ) chip_select = 0x01;
```

```
3994    else                                chip_select = 0x00;
3995
3996    error_code = SPI_Commit( chip_select );
3997    if ( error_code ) return error_code;
3998
3999    printf( "OK\n" );
4000    return SUCCESS;
4001 }
4002 /*****************************************************************************************************//**
4003  * @ingroup idi
4004  * @brief
4005  */
4006 static struct command_line  idi_cmd_spi[] =
4007 {
4008     {   NULL,   IDI_CMD__SPI_ID,                            "id",       "wishbone id: params:
    none"                                   },
4009     {   NULL,   IDI_CMD__SPI_Config_Get,                    "cfg",      "config dump: params:
    none"                                   },
4010     {   NULL,   IDI_CMD__SPI_Config_Clock_Hz,               "clk",      "clk:         params: [<clock freq in
    hertz>]"            },
4011     {   NULL,   IDI_CMD__SPI_Config_End_Cycle_Delay_Sec,    "ecd",      "end delay:   params: [<time in
    seconds>]"              },
4012     {   NULL,   IDI_CMD__SPI_Config_Mode,                   "mode",     "mode:        params:
    [<0/1/2/3>]"                            },
4013     {   NULL,   IDI_CMD__SPI_Config_SDI_Polarity,           "sdi",      "sdi pol:     params:
    [<true/1/false/0>]"             },
4014     {   NULL,   IDI_CMD__SPI_Config_SDO_Polarity,           "sdo",      "sdo pol:     params:
    [<true/1/false/0>]"             },
4015     {   NULL,   IDI_CMD__SPI_Config_SDIO_Wrap,              "wrap",     "sdo-->sdi:   params:
    [<true/1/false/0>]"             },
4016     {   NULL,   IDI_CMD__SPI_Config_Chip_Select_Behavior,   "csb",      "chip select behavior: params:
    [0/1/2/3/software/buffer/uint8_t/uint16_t]" },
4017     {   NULL,   IDI_CMD__SPI_Status,                        "status",   "status of both TX and RX
    buffers"                                },
4018     {   NULL,   IDI_CMD__SPI_Data,                          "data",     "read/write:  params: [one or more
    bytes/characters]"  },
4019     {   NULL,   IDI_CMD__SPI_FIFO,                          "fifo",     "fifo r/w:    params: [one or more
    bytes/characters]"  },
4020     {   NULL,   IDI_CMD__SPI_Commit,                        "commit"    "causes spi transactions to
    start"                          },
4021     {   NULL,   NULL,                                       NULL,       NULL
4022 };
4023 /*****************************************************************************************************//**
4024  * @ingroup idi
4025  * @brief
4026  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4027  *         code is returned.
4028  */
4029 int IDI_Command_Line_SPI( int argc, char* argv[] )
4030 {
4031     int         error_code;
4032     int         index;
4033     int         argc_new;
4034     char **     argv_new;
4035
4036     error_code = -EC_SYNTAX;
4037
4038     if ( argc < 1 ) return -EC_NOT_FOUND;
4039
4040     index = 0;
4041     while ( NULL != idi_cmd_spi[index].cmd_fnc )
4042     {
4043         if ( 0 == strcmpi( idi_cmd_spi[index].name, argv[0] )  )
4044         {
4045             argv_new = &(argv[1]);
4046             argc_new = argc - 1;
4047             if ( 0 == argc_new ) argv_new = NULL;
4048             error_code = (* idi_cmd_spi[index].cmd_fnc )( argc_new, argv_new );
4049             break;
4050         }
4051         index++;
4052     }
4053     return error_code;
4054 }
4055 /*****************************************************************************************************/
4056
4057
4058 /*****************************************************************************************************/
4059 /*****************************************************************************************************/
4060 /*****************************************************************************************************/
4061 /*    < < < <  F R A M  C O M M A N D  F U N C T I O N S  > > > >                                     */
4062
4063 /*****************************************************************************************************//**
4064  * @ingroup idi
4065  * @brief
4066  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4067  *         code is returned.
4068  */
4069 static int IDI_CMD__FRAM_Dump( int argc, char * argv[] )
4070 { /* idi fram dump <address> <length>  */
4071     uint16_t address;
4072     uint16_t length;
4073
```

```
4074    if ( argc < 1 ) return -EC_PARAMETER;
4075
4076    address = (uint16_t) strtol( argv[0], NULL, 0 );
4077
4078    if ( argc < 2 ) length = HEX_DUMP_BYTES_PER_LINE;
4079    else            length  = (uint16_t) strtol( argv[1], NULL, 0 );
4080
4081    if ( ( address + length ) > FRAM_DENSITY_BYTES ) length = FRAM_DENSITY_BYTES - address;
4082
4083    return FRAM_Report( address, length, stdout );
4084}
4085/*****************************************************************************************************//**
4086 * @ingroup idi
4087 * @brief
4088 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4089 *         code is returned.
4090 */
4091static int IDI_CMD__FRAM_Save( int argc, char * argv[] )
4092{ /* idi fram save <address> <length> <destination_file> */
4093    int         error_code;
4094    uint16_t    address;
4095    size_t      length;
4096    FILE *      out;
4097
4098    if ( argc < 3 ) return -EC_PARAMETER;
4099
4100    address = (uint16_t) strtol( argv[0], NULL, 0 );
4101    length  = (uint16_t) strtol( argv[1], NULL, 0 );
4102    out = fopen( argv[2], "w" );
4103    error_code = FRAM_Memory_To_File( address, length, out );
4104    fclose( out );
4105    return error_code;
4106}
4107/*****************************************************************************************************//**
4108 * @ingroup idi
4109 * @brief
4110 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4111 *         code is returned.
4112 */
4113static int IDI_CMD__FRAM_Load( int argc, char * argv[] )
4114{ /* idi fram load <address> <source_file> */
4115    int         error_code;
4116    uint16_t    address;
4117//  uint16_t    length;
4118    FILE *      out;
4119
4120    if ( argc < 2 ) return -EC_PARAMETER;
4121
4122    address = (uint16_t) strtol( argv[0], NULL, 0 );
4123    out = fopen( argv[2], "r" );
4124    error_code = FRAM_File_To_Memory( address, 0 /* no length specified */, out );
4125    fclose( out );
4126    return error_code;
4127}
4128/*****************************************************************************************************//**
4129 * @ingroup idi
4130 * @brief
4131 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4132 *         code is returned.
4133 */
4134static int IDI_CMD__FRAM_Init( int argc, char * argv[] )
4135{ /*  idi fram init <pattern list: 0x55 0x33 '3' '5' 'q' >  */
4136    int         error_code;
4137    int         index;
4138    uint8_t     buf[16];
4139    if ( argc < 1 )
4140    { /* initialize all zeros */
4141        error_code = FRAM_Set( 0, NULL );
4142    }
4143    else
4144    {
4145        if ( argc > 16 ) argc = 16;
4146        for ( index = 0; index < argc; index++ )
4147        {
4148            buf[index] = (uint8_t) strtol( argv[index], NULL, 0 );
4149        }
4150        error_code = FRAM_Set( argc, buf );
4151    }
4152    return error_code;
4153}
4154/*****************************************************************************************************//**
4155 * @ingroup idi
4156 * @brief
4157 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4158 *         code is returned.
4159 */
4160static int IDI_CMD__FRAM_WREN( int argc, char * argv[] )
4161{ /* idi fram WREN */
4162    (void)      argc;
4163    (void)      argv;
4164    return FRAM__Write_Enable_Latch_Set();
4165}
4166/*****************************************************************************************************//**
```

```
4167  * @ingroup idi
4168  * @brief
4169  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4170  *         code is returned.
4171  */
4172 static int IDI_CMD__FRAM_WRDI( int argc, char * argv[] )
4173 { /*  idi fram WRDI  */
4174     (void)      argc;
4175     (void)      argv;
4176     return FRAM__Write_Disable();
4177 }
4178 /*****************************************************************************************************//**
4179  * @ingroup idi
4180  * @brief
4181  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4182  *         code is returned.
4183  */
4184 static int IDI_CMD__FRAM_RDSR( int argc, char * argv[] )
4185 { /* idi fram RDSR  */
4186     int         error_code;
4187     uint8_t     status;
4188     (void)      argc;
4189     (void)      argv;
4190     error_code = FRAM__Read_Status_Register( &status );
4191     printf( "FRAM STATUS: 0x%02X\n", ((int) status) );
4192     return error_code;
4193 }
4194 /*****************************************************************************************************//**
4195  * @ingroup idi
4196  * @brief
4197  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4198  *         code is returned.
4199  */
4200 static int IDI_CMD__FRAM_WRSR( int argc, char * argv[] )
4201 { /*  idi fram WRSR <value>  */
4202 //  int     error_code;
4203     uint8_t status;
4204
4205     if ( argc < 1 ) return -EC_PARAMETER;
4206     status = (uint8_t) strtol( argv[0], NULL, 0 );
4207     return FRAM__Write_Status_Register( status );
4208 }
4209 /*****************************************************************************************************//**
4210  * @ingroup idi
4211  * @brief
4212  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4213  *         code is returned.
4214  */
4215 static int IDI_CMD__FRAM_RDID( int argc, char * argv[] )
4216 { /*  idi fram RDID  */
4217     uint32_t    id;
4218     (void)      argc;
4219     (void)      argv;
4220     FRAM__Read_ID( &id );
4221     printf( "FRAM ID: 0x%08X\n", id );
4222     return SUCCESS;
4223 }
4224 /*****************************************************************************************************//**
4225  * @ingroup idi
4226  * @brief
4227  */
4228 static struct command_line  idi_cmd_fram[] =
4229 {
4230     {   NULL,   IDI_CMD__FRAM_Dump,     "dump",     "params: <address> <length>"                          },
4231     {   NULL,   IDI_CMD__FRAM_Save,     "save"      "params: <address> <length> <binary destination file>"  },
4232     {   NULL,   IDI_CMD__FRAM_Load,     "load"      "params: <address> <binary source file name"          },
4233     {   NULL,   IDI_CMD__FRAM_Init,     "init"      "params: [byte/character] [byte/character] ..."       },
4234     {   NULL,   IDI_CMD__FRAM_WREN,     "wren"      "WRite Enable Latch Set"                              },
4235     {   NULL,   IDI_CMD__FRAM_WRDI,     "wrdi"      "WRite DIsable"                                       },
4236     {   NULL,   IDI_CMD__FRAM_RDSR,     "rdsr"      "ReaD Status Register"                                },
4237     {   NULL,   IDI_CMD__FRAM_WRSR,     "wrsr"      "WRite Status Register.  Params: <status>"            },
4238     {   NULL,   IDI_CMD__FRAM_RDID,     "rdid"      "ReaD ID Register"                                    },
4239     {   NULL,   NULL,                   NULL,       NULL                                                  },
4240 };
4241 /*****************************************************************************************************//**
4242  * @ingroup idi
4243  * @brief
4244  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4245  *         code is returned.
4246  */
4247 int IDI_Command_Line_FRAM( int argc, char* argv[] )
4248 {
4249     int         error_code;
4250     int         index;
4251     int         argc_new;
4252     char **     argv_new;
4253
4254     error_code = -EC_SYNTAX;
4255
4256     if ( argc < 1 ) return -EC_NOT_FOUND;
4257
4258     index = 0;
4259     while ( NULL != idi_cmd_fram[index].cmd_fnc )
```

```
4260    {
4261        if ( 0 == strcmpi( idi_cmd_fram[index].name, argv[0] )  )
4262        {
4263            argv_new = &(argv[1]);
4264            argc_new = argc - 1;
4265            if ( 0 == argc_new ) argv_new = NULL;
4266            error_code = (* idi_cmd_fram[index].cmd_fnc )( argc_new, argv_new );
4267            break;
4268        }
4269        index++;
4270    }
4271    return error_code;
4272 }
4273 /*****************************************************************************************************/
4274
4275
4276 /*****************************************************************************************************/
4277 /*****************************************************************************************************/
4278 /*****************************************************************************************************/
4279 /*    < < < <  D I G I T A L  I N P U T  C O M M A N D  F U N C T I O N S  > > > >                   */
4280
4281 /*****************************************************************************************************//**
4282  * @ingroup idi
4283  * @brief
4284  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4285  *         code is returned.
4286  */
4287 static int IDI_CMD__DIN_ID( int argc, char * argv[] )
4288 { /*  idi spi id */
4289     uint16_t    id;
4290     (void)      argc;
4291     (void)      argv;
4292     IDI_DIN_ID_Get( &id );
4293     printf( "DIN ID: 0x%04X\n", id );
4294     return SUCCESS;
4295 }
4296 /*****************************************************************************************************//**
4297  * @ingroup idi
4298  * @brief
4299  * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4300  *         code is returned.
4301  */
4302 static int IDI_CMD__DIN_All( int argc, char * argv[] )
4303 { /* idi din all [<binary/hex/group>]  */
4304     int     error_code;            /* used primarily for debug purposes */
4305     int     channel;
4306     //BOOL  value;
4307     int     cp;
4308     enum { MODE_NONE = 0, MODE_BINARY = 1, MODE_HEX = 2, MODE_ALL = 3 } mode_out;
4309     int     group;
4310     char    message[64];
4311     uint8_t din_grp[6];
4312     uint8_t mask;
4313     (void)  argc;
4314     (void)  argv;
4315
4316     mode_out = MODE_ALL;
4317     if ( argc > 0 )
4318     {
4319         int index;
4320         mode_out = MODE_NONE;
4321         for ( index = 0; index < argc; index++ )
4322         {
4323             if      ( 0 == strcmpi( "binary", argv[index] ) ) mode_out |= MODE_BINARY;
4324             else if ( 0 == strcmpi( "group",  argv[index] ) ) mode_out |= MODE_HEX;
4325             else if ( 0 == strcmpi( "hex",    argv[index] ) ) mode_out |= MODE_HEX;
4326             else                                              mode_out |= MODE_ALL;
4327         }
4328     }
4329     /* build in binary format */
4330     cp    = 0;
4331     group = 0;
4332     for ( group = 0; group < IDI_DIN_GROUP_QTY; group++ )
4333     {
4334         error_code = IDI_DIN_Group_Get( group, &(din_grp[group]) );
4335         mask = 0x01;
4336         for ( channel = 0; channel < 8; channel++ )
4337         {
4338             message[cp++] = !!(din_grp[group] & mask) ? '1' : '0';
4339             mask = mask << 1;
4340         }
4341         message[cp++] = ' ';
4342     }
4343     message[cp] = '\0';
4344     if ( MODE_BINARY == ( mode_out & MODE_BINARY ) )
4345     {
4346         printf( "DIN: %s\n", message );
4347     }
4348     if ( MODE_HEX == ( mode_out & MODE_HEX ) )
4349     {
4350         printf( "DIN:" );
4351         for ( group = 0; group < IDI_DIN_GROUP_QTY; group++ ) printf( " %02X", din_grp[group] );
4352         printf( "\n" );
```

```
4353    }
4354
4355    return SUCCESS;
4356}
4357/*****************************************************************************************//**
4358 * @ingroup idi
4359 * @brief
4360 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4361 *         code is returned.
4362 */
4363static int IDI_CMD__DIN_Channel( int argc, char * argv[] )
4364{
4365    int     error_code;            /* used primarily for debug purposes */
4366    int     channel;
4367    char    message[8];
4368    BOOL    value;
4369
4370    if ( argc < 1 ) return -EC_NOT_FOUND;
4371
4372    channel = (int) strtol( argv[0], NULL, 0 );
4373    error_code = IDI_DIN_Channel_Get( channel, &value );
4374    message[0] = value ? '1' : '0';
4375    message[1] = '\0';
4376
4377    printf( "DIN%02d: %s\n", channel, message );
4378    return SUCCESS;
4379}
4380/*****************************************************************************************//**
4381 * @ingroup idi
4382 * @brief
4383 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4384 *         code is returned.
4385 */
4386static int IDI_CMD__DIN_Group( int argc, char * argv[] )
4387{
4388    int     error_code; /* used primarily for debugging */
4389//  int     index;
4390    int     group;
4391    int     group_count;
4392    BOOL    do_all;
4393    uint8_t din_grp[6];
4394
4395    if ( argc < 1 ) do_all = true;
4396    else            do_all = false;
4397
4398    if ( 0 == strcmpi( "all", argv[0] ) ) do_all = true;
4399
4400    if ( do_all )
4401    { /* all */
4402        for ( group = 0; group < IDI_DIN_GROUP_QTY; group ++ )
4403        {
4404            error_code = IDI_DIN_Group_Get( group, &(din_grp[group]) );
4405        }
4406        group_count = IDI_DIN_GROUP_QTY;
4407    }
4408    else
4409    {
4410        group = (int) strtol( argv[0], NULL, 0 );
4411        error_code = IDI_DIN_Group_Get( group, &(din_grp[0]) );
4412        group_count = 1;
4413    }
4414
4415    printf( "DIN_GROUP:" );
4416    for ( group = 0; group < group_count; group++ )
4417    {
4418        printf( " 0x%02X", ((int) din_grp[group]) );
4419    }
4420    printf( "\n" );
4421    return SUCCESS;
4422}
4423/*****************************************************************************************//**
4424 * @ingroup idi
4425 * @brief
4426 */
4427static struct command_line  idi_cmd_din[] =
4428{
4429    {   NULL,   IDI_CMD__DIN_ID,        "id",       "params: none.  Reports the DIN board/component ID."    },
4430    {   NULL,   IDI_CMD__DIN_Channel,   "chan",     "params: <channel>"                                     },
4431    {   NULL,   IDI_CMD__DIN_Group,     "group",    "params: [<group_channel | all>]"                       },
4432    {   NULL,   IDI_CMD__DIN_All,       "all",      "reports all digital inputs in binary and hex"          },
4433    {   NULL,   NULL,                   NULL,       NULL                                                    },
4434};
4435/*****************************************************************************************//**
4436 * @ingroup idi
4437 * @brief
4438 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4439 *         code is returned.
4440 */
4441int IDI_Command_Line_Digital_Input( int argc, char* argv[] )
4442{ /*  idi din <channel/all>  */
4443    int         error_code;
4444    int         index;
4445    int         argc_new;
```

```
4446    char **     argv_new;
4447    char *      endptr;
4448    //int           channel;
4449
4450    if ( argc < 1 ) return -EC_NOT_FOUND;
4451
4452    error_code = -EC_SYNTAX;
4453
4454    //channel = (int) strtol( argv[0], &endptr, 0 );
4455    strtol( argv[0], &endptr, 0 ); /* just want to know where it fails */
4456    if ( argv[0] != endptr )
4457    {  /* assume channel number */
4458        error_code = (* idi_cmd_din[0].cmd_fnc )( argc, argv );
4459    }
4460    else
4461    { /* otherwise a normal command */
4462        index = 0;
4463        while ( NULL != idi_cmd_din[index].cmd_fnc )
4464        {
4465            if ( 0 == strcmpi( idi_cmd_din[index].name, argv[0] )  )
4466            {
4467                argv_new = &(argv[1]);
4468                argc_new = argc - 1;
4469                if ( 0 == argc_new ) argv_new = NULL;
4470                error_code = (* idi_cmd_din[index].cmd_fnc )( argc_new, argv_new );
4471                break;
4472            }
4473            index++;
4474        }
4475    }
4476    return error_code;
4477 }
4478 /*****************************************************************************************************/
4479
4480
4481 /*****************************************************************************************************/
4482 /*****************************************************************************************************/
4483 /*****************************************************************************************************/
4484 /*    < < < <  O T H E R   C O M M A N D   F U N C T I O N S  > > > >                                 */
4485
4486
4487 /*****************************************************************//**
4488  * @ingroup idi
4489  * @brief
4490  *
4491  * Either reads or writes a register using the form:
4492  * idi <register acronym> [<value>]
4493  *
4494  * If <value> is not include, then it is assumed to be a read.  If value is
4495  * included, then a write to the specified register is made.
4496  *
4497  * This function uses the definitions[] array which is global and built from
4498  * IDI_REGISTER_SET_DEFINITION macro which is a nicely organized register list.
4499  *
4500  * @param[in] argc  number of arguments including the executable file name
4501  * @param[in] argv  list of string arguments lex'd from the command line
4502  * @return SUCCESS (0) if no errors encountered, otherwise errors are reported
4503  * as a negative value.
4504  */
4505 int IDI_Command_Line_Register_Transaction( int argc, char* argv[] )
4506 {
4507     int error_code;
4508     int index;
4509     int found;
4510
4511     if ( argc < 1 ) return -EC_NOT_FOUND;
4512
4513     found = -1;
4514     index =  0;
4515     while ( definitions[index].direction != REG_DIR_NONE )
4516     {
4517         if ( 0 == strcmpi( definitions[index].acronym, argv[0] )  )
4518         {
4519             found = index;
4520             break;
4521         }
4522         index++;
4523     }
4524
4525     if ( found < 0 )
4526     {
4527         //printf( "ER: \n" );
4528         return  -EC_NOT_FOUND;
4529     }
4530
4531     if ( argc < 2 )
4532     { /* read operation */
4533         uint8_t value;
4534         error_code = IO_Read_U8( definitions[index].symbol, &value );
4535         if ( SUCCESS == error_code )
4536         {
4537             printf( "RD: %s=0x%02X\n",  definitions[index].acronym, value );
4538         }
```

```
4539    }
4540    else
4541    {
4542        uint8_t value;
4543        value = (uint8_t) strtol( argv[1], NULL, 0 );
4544        error_code = IO_Write_U8( definitions[index].symbol, value );
4545        if ( SUCCESS == error_code )
4546        {
4547            printf( "WR: %s=0x%02X\n", definitions[index].acronym, value );
4548        }
4549    }
4550
4551    return SUCCESS;
4552}
4553
4554/*****************************************************************************************************//**
4555 * @ingroup idi
4556 * @brief
4557 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4558 *         code is returned.
4559 */
4560static int IDI_CMD__Main_IO_Behavior( int argc, char * argv[] )
4561{ /*  idi spi ecd [<time sec>] */
4562
4563    if ( argc < 1 )
4564    { /* read */
4565        printf( "IO Simulate = %s\n", idi_dataset.io_simulate ? "true" : "false" );
4566        printf( "IO Report   = %s\n", idi_dataset.io_report   ? "true" : "false" );
4567    }
4568    else if ( argc > 1 )
4569    { /* write */
4570        if       ( 0 == strcmpi( "simulate", argv[0] ) )
4571        {
4572            idi_dataset.io_simulate = String_To_Bool( argv[1] );
4573        }
4574        else if  ( 0 == strcmpi( "report", argv[0] ) )
4575        {
4576            idi_dataset.io_report = String_To_Bool( argv[1] );
4577        }
4578        printf( "OK\n" );
4579    }
4580    else
4581    { /* read individual */
4582        if       ( 0 == strcmpi( "simulate", argv[0] ) )
4583        {
4584            printf( "IO Simulate = %s\n", idi_dataset.io_simulate ? "true" : "false" );
4585        }
4586        else if  ( 0 == strcmpi( "report", argv[0] ) )
4587        {
4588            printf( "IO Report   = %s\n", idi_dataset.io_report   ? "true" : "false" );
4589        }
4590    }
4591    return SUCCESS;
4592}
4593/*****************************************************************************************************//**
4594 * @ingroup idi
4595 * @brief
4596 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4597 *         code is returned.
4598 */
4599static int IDI_CMD__Main_Base( int argc, char * argv[] )
4600{ /*  idi spi ecd [<time sec>] */
4601
4602    if ( argc < 1 )
4603    { /* read */
4604        printf( "base        = 0x%04X\n", idi_dataset.base_address );
4605    }
4606    else
4607    { /* write */
4608        idi_dataset.base_address = (uint16_t) strtol( argv[0], NULL, 0 );
4609        printf( "OK\n" );
4610    }
4611    return SUCCESS;
4612}
4613/*****************************************************************************************************//**
4614 * @ingroup idi
4615 * @brief
4616 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4617 *         code is returned.
4618 */
4619static int IDI_CMD__Main_Irq_Number( int argc, char * argv[] )
4620{ /*  idi spi ecd [<time sec>] */
4621
4622    if ( argc < 1 )
4623    { /* read */
4624#if defined( __MSDOS__ )
4625        printf( "irq         = %u\n", idi_dataset.irq_number );
4626#else
4627        printf( "irq         = %u\n", idi_dataset.irq_number );
4628#endif
4629    }
4630    else
4631    { /* write */
```

```
4632        idi_dataset.irq_number = (unsigned int) strtol( argv[0], NULL, 0 );
4633        printf( "OK\n" );
4634    }
4635    return SUCCESS;
4636}
4637/*****************************************************************************************************//**
4638 * @ingroup idi
4639 * @brief
4640 * Number of interrupt cycles.  If "loop" is used, then the count can be terminated by
4641 * pressing any key on the keyboard input.
4642 *
4643 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4644 *         code is returned.
4645 */
4646static int IDI_CMD__Main_I_Count( int argc, char * argv[] )
4647{ /*  idi spi ecd [<time sec>] */
4648
4649    if ( argc < 1 )
4650    { /* read */
4651#if defined( __MSDOS__ )
4652        printf( "iqty        = %u\n", idi_dataset.irq_quantity );
4653#else
4654        printf( "iqty        = %lu\n", idi_dataset.irq_quantity );
4655#endif
4656    }
4657    else
4658    { /* write */
4659        idi_dataset.irq_quantity = (size_t) strtol( argv[0], NULL, 0 );
4660        printf( "OK\n" );
4661    }
4662    return SUCCESS;
4663}
4664/*****************************************************************************************************//**
4665 * @ingroup idi
4666 * @brief
4667 */
4668static struct command_line  idi_cmd_set[] =
4669{
4670    {   NULL,   IDI_CMD__Main_Base,             "base",     "params: [<address>]"                           },
4671    {   NULL,   IDI_CMD__Main_IO_Behavior,      "io",       "params: [<simulate>/<report>]"                 },
4672    {   NULL,   IDI_CMD__Main_Irq_Number,       "irq",      "params: [<irq_number 0 to 15>]"                },
4673    {   NULL,   IDI_CMD__Main_I_Count,          "iqty",     "params: [<number>].  Number of interrupts."    },
4674    {   NULL,   NULL,                           NULL,       NULL                                            },
4675};
4676/*****************************************************************************************************//**
4677 * @ingroup idi
4678 * @brief
4679 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
4680 *         code is returned.
4681 */
4682int IDI_Command_Line_Set( int argc, char* argv[] )
4683{
4684    int         error_code;
4685    int         index;
4686    int         argc_new;
4687    char **     argv_new;
4688
4689    error_code = -EC_SYNTAX;
4690
4691    if ( argc < 1 )
4692    {
4693        index = 0;
4694        while ( NULL != idi_cmd_set[index].cmd_fnc )
4695        {
4696            argv_new = &(argv[1]);
4697            argc_new = argc - 1;
4698            if ( 0 == argc_new ) argv_new = NULL;
4699            error_code = (* idi_cmd_set[index].cmd_fnc )( argc_new, argv_new );
4700            index++;
4701        }
4702    }
4703    else
4704    {
4705        index = 0;
4706        while ( NULL != idi_cmd_set[index].cmd_fnc )
4707        {
4708            if ( 0 == strcmpi( idi_cmd_set[index].name, argv[0] )  )
4709            {
4710                argv_new = &(argv[1]);
4711                argc_new = argc - 1;
4712                if ( 0 == argc_new ) argv_new = NULL;
4713                error_code = (* idi_cmd_set[index].cmd_fnc )( argc_new, argv_new );
4714                break;
4715            }
4716            index++;
4717        }
4718    }
4719    return error_code;
4720}
4721/*****************************************************************************************************//**
4722 * @ingroup idi
4723 * @brief
4724 * @return A zero (SUCCESS) is returned if successful, otherwise a negative error
```

```
 *         code is returned.
 */
int IDI_Command_Line_Dump( int argc, char* argv[] )
{
    int    error_code;
    FILE * fd_out;

    if ( argc > 0 )
    {
        fd_out = fopen( argv[0], "w" );
        if ( NULL == fd_out ) fd_out = stdout;
    }
    else
    {
        fd_out = stdout;
    }
    error_code = IDI_Register_Report_CSV( definitions, fd_out );

    if ( (argc > 0) && (NULL != fd_out) && (stdout != fd_out) )
    {
        fclose( fd_out );
    }
    return error_code;
}

/*****************************************************************************************************//**
 * @ingroup idi
 * @brief
 */
static struct command_line  idi_cmd_top[] =
{
    {   NULL,           IDI_Command_Line_Dump,          "dump",     "Register information in CSV format"            },
    {   idi_cmd_set,    IDI_Command_Line_Set,           "set",      "Set/Get main parameters"                       },
    {   idi_cmd_spi,    IDI_Command_Line_SPI,           "spi",      "SPI related functions"                         },
    {   idi_cmd_fram,   IDI_Command_Line_FRAM,          "fram",     "FRAM related functions"                        },
    {   idi_cmd_din,    IDI_Command_Line_Digital_Input, "din",      "Digital input related functions"               },
    {   NULL,           NULL,                           NULL,       NULL                                            },
};
/*****************************************************************//**
 * @ingroup idi
 * @brief
 * Processes and dispatches the top level of the command and passes the remaining
 * string list onto specialized functions to further process arguments.
 * If no command is specified then a help output is produced.
 *
 * @param[in] argc  number of arguments including the executable file name
 * @param[in] argv  list of string arguments lex'd from the command line
 * @return SUCCESS (0) if no errors encountered, otherwise errors are reported
 * as a negative value.
 */
int IDI_Command_Line_Main( int argc, char* argv[] )
{
    int     error_code;
    int     index;
    BOOL    not_found;
    int     argc_new;
    char ** argv_new;


    error_code = -EC_SYNTAX;

    if ( argc < 1 ) return -EC_NOT_FOUND;

    /* there has to be at least one argument at this point */
    not_found = true;
    index     = 0;
    while ( NULL != idi_cmd_top[index].cmd_fnc )
    {
        if ( 0 == strcmpi( idi_cmd_top[index].name, argv[0] )  )
        {
            not_found = false;
            argv_new = &(argv[1]);
            argc_new = argc - 1;
            if ( 0 == argc_new ) argv_new = NULL;
            error_code = (* idi_cmd_top[index].cmd_fnc )( argc_new, argv_new );
            if ( error_code ) goto IDI_COMMAND_LINE_MAIN_TERMINATE;
            break;
        }
        index++;
    }

    if ( not_found )
    { /* assume that it is a register related transaction */
        argv_new = &(argv[0]);
        argc_new = argc - 0;
        error_code = IDI_Command_Line_Register_Transaction( argc_new, argv_new );
        if ( error_code ) goto IDI_COMMAND_LINE_MAIN_TERMINATE;
    }
IDI_COMMAND_LINE_MAIN_TERMINATE:
    return error_code;
}

/*****************************************************************************************************//**
```

```
4818  * @ingroup idi
4819  * @brief
4820  */
4821 static int IDI_Command_Line_Prefix_Irq( int   argc,    /* argument index in this case   */
4822                                         char* argv[] /* always null in this case       */
4823                                       )
4824 {
4825     (void)  argc;
4826     (void)  argv;
4827     idi_dataset.irq_please_install_handler_request = true;
4828 //  printf( "OK on irq_please_install_handler_request\n" );
4829     return SUCCESS;
4830 }
4831 /*****************************************************************************************//**
4832  * @ingroup idi
4833  * @brief
4834  */
4835 static int IDI_Command_Line_Prefix_Loop( int   argc,    /* argument index in this case */
4836                                          char* argv[]   /* always null in this case */
4837                                        )
4838 {
4839     (void)  argc;
4840     (void)  argv;
4841     idi_dataset.loop_command = true;
4842 //  printf( "OK on loop_command\n" );
4843     return SUCCESS;
4844 }
4845 /*****************************************************************************************//**
4846  * @ingroup idi
4847  * @brief
4848  */
4849 static struct command_line  idi_cmd_prefix[] =
4850 {
4851     {   NULL,   IDI_Command_Line_Prefix_Irq,            "irq",      "allows commands to handle interrupts"              },
4852     {   NULL,   IDI_Command_Line_Prefix_Loop,           "loop",     "any command loops until key pressed"               },
4853     {   NULL,   NULL,                                   NULL,       NULL                                                },
4854 };
4855 /*****************************************************************************************//**
4856  * @ingroup idi
4857  * @brief
4858  * @return Index value to next argument in the list (i.e. updated index), or negative value
4859  *         indicating an error condition.
4860  */
4861 int IDI_Command_Line_Prefix( int argc, char* argv[] )
4862 { /*  idi din <channel/all>  */
4863     int         error_code;
4864     int         ti;     /* table index      */
4865     int         ait;    /* argument index test */
4866     int         ai;     /* argument index   */
4867     BOOL        not_found;
4868
4869     if ( argc < 1 ) return 0;
4870
4871 #if (0)
4872     /* prefixed portion of the commands */
4873     if ( 0 == strcmpi( "irq", argv[index]) )
4874     { /* invoke interrupts for this session */
4875         idi_dataset.irq_please_install_handler_request = true;
4876         index++;
4877     }
4878
4879     if ( 0 == strcmpi( "loop", argv[index]) )
4880     { /* invoke interrupts for this session */
4881         idi_dataset.loop_command = true;
4882         index++;
4883     }
4884 #endif
4885
4886     ai = 0;
4887     for ( ait = 0; ait < argc; ait++ )
4888     {
4889         not_found = true;
4890         ti        = 0;
4891         while ( NULL != idi_cmd_prefix[ti].cmd_fnc )
4892         {
4893             if ( 0 == strcmpi( idi_cmd_prefix[ti].name, argv[ait] )  )
4894             {
4895                 not_found = false;
4896                 error_code = (* idi_cmd_prefix[ti].cmd_fnc )( ti, NULL );
4897                 if ( error_code < 0 ) return error_code;
4898                 else                  ai++;
4899                 break;
4900             }
4901             ti++;
4902         }
4903         if ( not_found ) return ai;
4904     }
4905     return ai;
4906 }
4907
4908 /*****************************************************************************************/
4909 /*****************************************************************************************/
4910 /*****************************************************************************************/
```

```
4911
4912
4913
4914
4915 /*****************************************************************************************/
4916 /*****************************************************************************************/
4917 /*****************************************************************************************/
4918 /*                  < < < <  M A I N   F U N C T I O N S  > > > >                        */
4919
4920 /*************************************************************************//**
4921  * @ingroup idi
4922  * @brief
4923  * Outputs a help listing to the user.
4924  */
4925 void IDI_Help( FILE * out )
4926 {
4927     struct command_line * clt; /* command line table pointer */
4928     int index;
4929     int top;
4930
4931     top = 0;
4932     fprintf( out, "\n" );
4933     fprintf( out, "Isolated Digital Input Test Code\n" );
4934     fprintf( out, "Apex Embedded Systems\n" );
4935     fprintf( out, "Revision: %s\n", idi_dataset.svn_revision_string );
4936
4937     fprintf( out, "\n" );
4938     fprintf( out, "Examples:\n" );
4939     fprintf( out, "    idi dig0          <-- reports the digital input group 0 port.\n" );
4940     fprintf( out, "    idi loop dig0     <-- loops until key pressed\n" );
4941     fprintf( out, "    idi irq loop dig0 <-- loops while counting interrupts or until iqty reached\n" );
4942     fprintf( out, "    idi irq dig0      <-- does command once and uses interrupts for short time\n" );
4943
4944     fprintf( out, "\n" );
4945     fprintf( out, "help - outputs help information\n" );
4946
4947     top = 0;
4948     while ( NULL != idi_cmd_prefix[top].name )
4949     {
4950         fprintf( out, "\n" );
4951         fprintf( out, "%-4s - %s\n", idi_cmd_prefix[top].name, idi_cmd_prefix[top].help );
4952         top++;
4953     }
4954
4955 #if(0)
4956 //to remove
4957     fprintf( out, "\n" );
4958     fprintf( out, "irq  - allows subsequent command to handle interrupts\n" );
4959
4960     fprintf( out, "\n" );
4961     fprintf( out, "loop - any command below can run in a loop until key pressed\n" );
4962 #endif
4963     top = 0;
4964     while ( NULL != idi_cmd_top[top].name )
4965     {
4966         fprintf( out, "\n" );
4967         fprintf( out, "%-4s - %s\n", idi_cmd_top[top].name, idi_cmd_top[top].help );
4968         index = 0;
4969         if ( NULL != idi_cmd_top[top].link )
4970         {
4971             clt = idi_cmd_top[top].link;
4972             while ( NULL != clt[index].help )
4973             {
4974                 fprintf( out, "       %8s - %s\n", clt[index].name, clt[index].help );
4975                 index++;
4976             }
4977         }
4978         top++;
4979     }
4980     fprintf( out, "\n" );
4981 }
4982 /*************************************************************************//**
4983  * @ingroup idi
4984  * @brief
4985  * Runs upon application exit.  It saves the idi_dataset data structure.
4986  *
4987  * @return SUCCESS (0) if no errors encountered, otherwise errors are reported
4988  * as a negative value.
4989  */
4990 int IDI_Termination( void )
4991 {
4992     FILE * fd;
4993
4994 #if defined( __MSDOS__ )
4995     IOKern_Resource_Termination();
4996 #endif
4997
4998     if ( idi_dataset.irq_handler_active )
4999     {
5000 #if defined( __MSDOS__ )
5001         printf( " --> irq %u: count = %u\n", idi_dataset.irq_number, idi_dataset.irq_count );
5002 #else
5003         printf( " --> irq %u: count = %lu\n", idi_dataset.irq_number, idi_dataset.irq_count );
```

```
#endif
    }
    idi_dataset.irq_handler_active = false;
    idi_dataset.irq_please_install_handler_request = false;

    /* save the data set */
    fd = fopen( "idi_init.bin", "w" );
    if ( NULL == fd )
    { /* defaults */
        return -EC_INIT_FILE;
    }
    else
    { /* read in dataset */
        fwrite( &idi_dataset, 1, sizeof( struct idi_dataset ), fd );
        fclose( fd );
    }
    return SUCCESS;
}
/*****************************************************************//**
 * @ingroup idi
 * @brief
 * Runs upon application startup.  It restores the idi_dataset data structure
 * or if the file cannot be found it will simply initialize those parameters
 * to default values.
 *
 * @return SUCCESS (0) if no errors encountered, otherwise errors are reported
 * as a negative value.
 */
int IDI_Initialization( void )
{
    FILE * fd;

    /* restore the data set, if we can otherwise initialize with defaults */
    fd = fopen( "idi_init.bin", "r" );
    if ( NULL == fd )
    { /* defaults */
        memset( &idi_dataset, 0, sizeof(struct idi_dataset) );
        idi_dataset.base_address        = 0xff00;
        idi_dataset.bank_previous       = IDI_BANK_0;
    }
    else
    { /* read in dataset */
        fread( &idi_dataset, 1, sizeof( struct idi_dataset ), fd );
        fclose( fd );
    }
    idi_dataset.svn_revision_string = idi_svn_revision_string;
    idi_dataset.irq_handler_active = false;
    idi_dataset.irq_please_install_handler_request = false;
    idi_dataset.irq_count          = 0;
    idi_dataset.irq_count_previous = 0;
    idi_dataset.loop_command       = false;
#if defined( __MSDOS__ )
    IOKern_Resource_Initialization();
#endif

    return SUCCESS;
}


/*****************************************************************//**
 * @ingroup idi
 * @brief
 * Processes and dispatches the top level of the command and passes the remaining
 * string list onto specialized functions to further process arguments.
 * If no command is specified then a help output is produced.
 *
 * @param[in] argc  number of arguments including the executable file name
 * @param[in] argv  list of string arguments lex'd from the command line
 * @return SUCCESS (0) if no errors encountered, otherwise errors are reported
 * as a negative value.
 */
int main( int argc, char* argv[] )
{
    //int index;
    //int count;
    int     error_code;
    int     index;
    int     argc_new;
    char ** argv_new;

/* used only for Win7 debugging sessions with cygwin */
#if(1)
    setvbuf(stdout, NULL, _IONBF, 0);
    setvbuf(stderr, NULL, _IONBF, 0);
#endif

    error_code = IDI_Initialization();
    if ( error_code ) goto Main_Termination;
//      printf( "Hello\n");
//      return SUCCESS;

#if(0)
    count = argc;
```

```
5097    index = 0;     /* OK, zero value is name of executable file  */
5098    while ( count > 0 )
5099    {
5100        printf( "index = %d, str = <%s>\n", index, argv[index] );
5101        index++;
5102        count--;
5103    }
5104#endif
5105
5106    index = 1;
5107    if ( argc > 1 )
5108    {
5109        if ( 0 == strcmpi( "help", argv[index]) )
5110        {
5111            IDI_Help( stdout );
5112            goto Main_Termination;
5113        }
5114
5115        /* prefixed portion of the commands */
5116        argv_new = &(argv[index]);
5117        argc_new = argc - index;
5118        index   += IDI_Command_Line_Prefix( argc_new, argv_new );
5119
5120        do
5121        { /* assumes that all functions utilize arguments in read only fashion */
5122            argv_new = &(argv[index]);
5123            argc_new = argc - index;
5124            error_code = IDI_Command_Line_Main( argc_new, argv_new );
5125            if ( error_code ) goto Main_Termination_Error_Codes;
5126
5127            if ( idi_dataset.irq_handler_active )
5128            {
5129                if ( idi_dataset.irq_count != idi_dataset.irq_count_previous )
5130                {
5131//TODO: spinlock - SPIN_LOCK_IRQ_SAVE method
5132                    idi_dataset.irq_count_previous = idi_dataset.irq_count;
5133//TODO: release spinlock
5134#if defined( __MSDOS__ )
5135                    printf( " --> irq %u: count = %u\n", idi_dataset.irq_number, idi_dataset.irq_count );
5136#else
5137                    printf( " --> irq %u: count = %lu\n", idi_dataset.irq_number, idi_dataset.irq_count );
5138#endif
5139                    if ( idi_dataset.irq_count_previous >= idi_dataset.irq_quantity )
5140                    {
5141                        break; /* quit the loop */
5142                    }
5143                }
5144            }
5145        } while ( ( idi_dataset.loop_command ) && !Character_Get(NULL) );
5146
5147#if(0)
5148        if ( 0 == strcmpi( "loop", argv[index]) )
5149        { /* loop until key is pressed */
5150            if ( argc > 2 )
5151            {
5152                index++;
5153
5154            }
5155        }
5156        else
5157        { /* non-looping behavior */
5158            argv_new = &(argv[index]);
5159            argc_new = argc - index;
5160            error_code = IDI_Command_Line_Main( argc_new, argv_new );
5161            if ( error_code ) goto Main_Termination_Error_Codes;
5162        }
5163#endif
5164
5165    }
5166    else
5167    { /* produce help */
5168        IDI_Help( stdout );
5169        goto Main_Termination;
5170    }
5171
5172Main_Termination:
5173    IDI_Termination();
5174    return error_code;
5175
5176Main_Termination_Error_Codes:
5177    IDI_Termination();
5178    printf( "ERROR: %d, %s\n",  error_code, EC_Code_To_Human_Readable( error_code ) );
5179    return error_code;
5180}
5181
5182
5183
5184
```