Isolated Digital Input Test Utility

Generated by Doxygen 1.8.8

Thu Mar 5 2015 09:18:50

Contents

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 command_line Struct Reference

Data Fields

- int(∗ cmd_fnc)(int argc, char ∗argv[])
- char ∗ name
- char ∗ help

3.1.1 Detailed Description

Definition at line 2440 of file idi.c.

3.1.2 Field Documentation

3.1.2.1 int(∗ cmd_fnc)(int argc, char ∗argv[])

Definition at line 2442 of file idi.c.

Referenced by IDI_Command_Line_Digital_Input(), IDI_Command_Line_FRAM(), IDI_Command_Line_Set(), and IDI_Command_Line_SPI().

3.1.2.2 char∗ help

Definition at line 2444 of file idi.c.

Referenced by IDI_Help().

3.1.2.3 char∗ name

Definition at line 2443 of file idi.c.

Referenced by IDI_Command_Line_Digital_Input(), IDI_Command_Line_FRAM(), IDI_Command_Line_Set(), IDI_Command_Line_SPI(), and IDI_Help().

The documentation for this struct was generated from the following file:

- src/idi.c

3.2 din_cfg Struct Reference

Data Fields

- struct {
  BOOL falling_edge
  BOOL interrupt_enable
  } chan [IDI_DIN_QTY]

3.2.1 Detailed Description

Definition at line 359 of file idi.c.

3.2.2 Field Documentation

3.2.2.1 struct { ... } chan[IDI_DIN_QTY]

3.2.2.2 BOOL falling_edge

Definition at line 363 of file idi.c.

3.2.2.3 BOOL interrupt_enable

Definition at line 364 of file idi.c.

The documentation for this struct was generated from the following file:

- src/idi.c

3.3 ec_human_readable Struct Reference

Data Fields

- EC_ENUM error_code
- const char * message

3.3.1 Detailed Description

Definition at line 172 of file idi.c.

3.3.2 Field Documentation

3.3.2.1 EC_ENUM error_code

Definition at line 174 of file idi.c.

3.3.2.2 const char* message

Definition at line 175 of file idi.c.

Referenced by EC_Code_To_Human_Readable().

The documentation for this struct was generated from the following file:

- src/idi.c

3.4 idi_dataset Struct Reference

Data Fields

- struct din_cfg din_cfg
- struct spi_cfg spi_cfg
- uint8_t bank_previous
- uint16_t base_address
- uint16_t spi_id
- BOOL io_simulate
- BOOL io_report
- const char * svn_revision_string
- uint8_t fram_block [FRAM_BLOCK_SIZE]
- uint8_t spi_block [SPI_BLOCK_SIZE]
- char * message [IDI_MESSAGE_SIZE]

3.4.1 Detailed Description

Definition at line 376 of file idi.c.

3.4.2 Field Documentation

3.4.2.1 uint8_t bank_previous

Definition at line 380 of file idi.c.

Referenced by IDI_Initialization(), IO_Read_U8(), and IO_Write_U8().

3.4.2.2 uint16_t base_address

Definition at line 381 of file idi.c.

Referenced by IDI_CMD__Main_Base(), IDI_Initialization(), IO_Read_U8(), and IO_Write_U8().

3.4.2.3 struct din_cfg din_cfg

Definition at line 378 of file idi.c.

3.4.2.4 uint8_t fram_block[FRAM_BLOCK_SIZE]

Definition at line 388 of file idi.c.

Referenced by FRAM_File_To_Memory(), FRAM_Memory_To_File(), FRAM_Report(), and FRAM_Set().

3.4.2.5 BOOL io_report

Definition at line 384 of file idi.c.

Referenced by IDI_CMD__Main_IO_Behavior(), IO_Read_U8(), and IO_Write_U8().

3.4.2.6 BOOL io_simulate

Definition at line 383 of file idi.c.

Referenced by IDI_CMD__Main_IO_Behavior(), IO_Read_U8(), and IO_Write_U8().

3.4.2.7 char* message[IDI_MESSAGE_SIZE]

Definition at line 394 of file idi.c.

Referenced by EC_Code_To_Human_Readable().

3.4.2.8 uint8_t spi_block[SPI_BLOCK_SIZE]

Definition at line 391 of file idi.c.

3.4.2.9 struct spi_cfg spi_cfg

Definition at line 379 of file idi.c.

Referenced by SPI_Configuration_Get(), and SPI_Configuration_Set().

3.4.2.10 uint16_t spi_id

Definition at line 382 of file idi.c.

3.4.2.11 const char* svn_revision_string

Definition at line 385 of file idi.c.

Referenced by IDI_Help(), and IDI_Initialization().

The documentation for this struct was generated from the following file:

- src/idi.c

3.5 reg_definition Struct Reference

Data Fields

- IDI_REG_ENUM symbol
- REG_DIR_ENUM direction
- IDI_BANK_ENUM bank
- uint16_t physical_offset
- char * symbol_name
- char * acronym

3.5.1 Detailed Description

Definition at line 302 of file idi.c.

3.5.2 Field Documentation

3.5.2.1 char* acronym

Definition at line 309 of file idi.c.

3.5.2.2 IDI_BANK_ENUM bank

Definition at line 306 of file idi.c.

3.5.2.3 REG_DIR_ENUM direction

Definition at line 305 of file idi.c.

3.5.2.4 uint16_t physical_offset

Definition at line 307 of file idi.c.

3.5.2.5 IDI_REG_ENUM symbol

Definition at line 304 of file idi.c.

3.5.2.6 char* symbol_name

Definition at line 308 of file idi.c.

Referenced by IO_Get_Symbol_Name().

The documentation for this struct was generated from the following file:

- src/idi.c

3.6 spi_cfg Struct Reference

Data Fields

- BOOL sdio_wrap
- BOOL sdo_polarity
- BOOL sdi_polarity
- BOOL sclk_phase
- BOOL sclk_polarity
- SPI_CSB_ENUM chip_select_behavior
- uint8_t end_cycle_delay
- uint16_t half_clock_interval
- double clock_hz
- double end_delay_ns

3.6.1 Detailed Description

Definition at line 327 of file idi.c.

3.6.2 Field Documentation

3.6.2.1 SPI_CSB_ENUM chip_select_behavior

CSB[2:0]

Definition at line 334 of file idi.c.

Referenced by IDI_CMD__SPI_Config_Chip_Select_Behavior(), SPI_Configuration_Get(), SPI_Configuration_Initialize(), SPI_Configuration_Set(), and SPI_Report_Configuration_Text().

3.6.2.2 double clock_hz

if nonzero, code will compute half_clock_interval

Definition at line 338 of file idi.c.

Referenced by IDI_CMD__SPI_Config_Clock_Hz(), SPI_Configuration_Get(), SPI_Configuration_Initialize(), SPI_Configuration_Set(), and SPI_Report_Configuration_Text().

3.6.2.3 uint8_t end_cycle_delay

ECD[7:0]

Definition at line 335 of file idi.c.

Referenced by SPI_Configuration_Get(), SPI_Configuration_Initialize(), SPI_Configuration_Set(), and SPI_Report_Configuration_Text().

3.6.2.4 double end_delay_ns

if nonzero, code will compute end_cycle_dealy

Definition at line 339 of file idi.c.

Referenced by IDI_CMD__SPI_Config_End_Cycle_Delay_Sec(), SPI_Configuration_Get(), SPI_Configuration_↩ Initialize(), SPI_Configuration_Set(), and SPI_Report_Configuration_Text().

3.6.2.5 uint16_t half_clock_interval

HCI[11:0]

Definition at line 336 of file idi.c.

Referenced by IDI_CMD__SPI_Config_End_Cycle_Delay_Sec(), SPI_Configuration_Get(), SPI_Configuration_↩ Initialize(), SPI_Configuration_Set(), and SPI_Report_Configuration_Text().

3.6.2.6 BOOL sclk_phase

SCLK_PHA

Definition at line 332 of file idi.c.

Referenced by IDI_CMD__SPI_Config_Mode(), SPI_Configuration_Get(), SPI_Configuration_Initialize(), SPI_↩ Configuration_Set(), and SPI_Report_Configuration_Text().

3.6.2.7 BOOL sclk_polarity

SCLK_POL

Definition at line 333 of file idi.c.

Referenced by IDI_CMD__SPI_Config_Mode(), SPI_Configuration_Get(), SPI_Configuration_Initialize(), SPI_↩ Configuration_Set(), and SPI_Report_Configuration_Text().

3.6.2.8 BOOL sdi_polarity

SDI_POL

Definition at line 331 of file idi.c.

Referenced by IDI_CMD__SPI_Config_SDI_Polarity(), SPI_Configuration_Get(), SPI_Configuration_Initialize(), SPI_↩ Configuration_Set(), and SPI_Report_Configuration_Text().

3.6.2.9 BOOL sdio_wrap

SDIO_WRAP

Definition at line 329 of file idi.c.

Referenced by IDI_CMD__SPI_Config_SDIO_Wrap(), SPI_Configuration_Get(), SPI_Configuration_Initialize(), and S↩ PI_Configuration_Set().

3.6.2.10 BOOL sdo_polarity

SDO_POL

Definition at line 330 of file idi.c.

Referenced by IDI_CMD__SPI_Config_SDO_Polarity(), SPI_Configuration_Get(), SPI_Configuration_Initialize(), and SPI_Configuration_Set().

The documentation for this struct was generated from the following file:

- src/idi.c

3.7 spi_status Struct Reference

Data Fields

- BOOL tx_status
- BOOL full
- BOOL empty
- int fifo_size
- int fifo_count

3.7.1 Detailed Description

Definition at line 342 of file idi.c.

3.7.2 Field Documentation

3.7.2.1 BOOL empty

Definition at line 346 of file idi.c.

Referenced by SPI_Report_Status_Text(), SPI_Status_Read(), and SPI_Status_Write().

3.7.2.2 int fifo_count

Definition at line 348 of file idi.c.

Referenced by SPI_Report_Status_Text(), SPI_Status_Read(), and SPI_Status_Write().

3.7.2.3 int fifo_size

Definition at line 347 of file idi.c.

Referenced by SPI_Report_Status_Text(), SPI_Status_Read(), and SPI_Status_Write().

3.7.2.4 BOOL full

Definition at line 345 of file idi.c.

Referenced by SPI_Report_Status_Text(), SPI_Status_Read(), and SPI_Status_Write().

3.7.2.5 BOOL tx_status

Definition at line 344 of file idi.c.

Referenced by SPI_Report_Status_Text(), SPI_Status_Read(), and SPI_Status_Write().

The documentation for this struct was generated from the following file:

- src/idi.c

Chapter 4

File Documentation

4.1 src/idi.c File Reference

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

Data Structures

- struct ec_human_readable
- struct reg_definition
- struct spi_cfg
- struct spi_status
- struct din_cfg
- struct idi_dataset
- struct command_line

Macros

- #define strcmpi strcasecmp
- #define IDI_REV "$Date: 2015-03-05 08:48:57 -0600 (Thu, 05 Mar 2015) $"

  The Subversion (SVN) time/date marker which is updated during commit of the source file to the repository.
- #define false 0
- #define true 1
- #define CLOCK_PERIOD_SEC 20.0e-9

  Board clock period as defined by the on-board oscillator which is 50MHz.
- #define ID_ALWAYS_REPORT_AS_GOOD 1
- #define IDI_ERROR_CODES(_)

  An organized error code listing. This macro is used to build the complete error code enumeration list.
- #define EC_EXTRACT_ENUM(symbol, code, message) symbol = code,
- #define EC_EXTRACT_HUMAN_READABLE(symbol, code, message) { code, message },
- #define EC_HUMAN_READABLE_TERMINATE { 0, NULL }
- #define IDI_REGISTER_SET_DEFINITION(_)

Organized list of registers and associate attributes of each of the registers. This macro does not consume any memory of in itself, and is only 'consumed' or used to automatically build enumerations and pre-built data structures.

- #define REG_BANK_SET(bank) (bank << 8)
- #define REG_OFFSET_SET(offset) (offset & 0xff)
- #define REG_LOCATION_BANK_GET(location) ((location >> 8) & 0xFF)
- #define REG_LOCATION_OFFSET_GET(location) (location & 0xff)
- #define REG_EXTRACT_ENUM(symbol, index, offset, register_bytes, aperture_bytes, read_write, name, bank) symbol = (REG_BANK_SET(bank) | REG_OFFSET_SET(offset)),
- #define REG_EXTRACT_DEFINITION(symbol, index, offset, register_bytes, aperture_bytes, read_write, name, bank) { ((IDI_REG_ENUM) (REG_BANK_SET(bank) | REG_OFFSET_SET(offset))), read_write, bank, offset, #symbol, name },
- #define IDI_DIN_GROUP_SIZE 8
- #define IDI_DIN_SHIFT_RIGHT 3
- #define IDI_DIN_GROUP_QTY 6
- #define IDI_DIN_QTY (IDI_DIN_GROUP_SIZE * IDI_DIN_GROUP_QTY)
- #define FRAM_BLOCK_SIZE 256
- #define IDI_MESSAGE_SIZE 256
- #define SPI_BLOCK_SIZE 256
- #define IDI_IO_DIRECTION_TEST 1

Typedefs

- typedef int BOOL

  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.

Enumerations

- enum { FALSE = 0, TRUE = 1 }
- enum { ID_DIN = 0x8012, ID_SPI = 0x8013 }

  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.
- enum EC_ENUM
- enum IDI_BANK_ENUM {
  IDI_BANK_0 = 0x00, IDI_BANK_1 = 0x40, IDI_BANK_2 = 0x80, IDI_BANK_3 = 0xC0,
  IDI_BANK_4 = 0x20, IDI_BANK_5 = 0x60, IDI_BANK_6 = 0xA0, IDI_BANK_7 = 0xE0,
  IDI_BANK_NONE = 0xFE, IDI_BANK_UNDEFINED = 0xFF }

  The bank register mapping. This mapping is upwardly compatible with the legacy hardware banking register. It also allows for future expansion utilizing the lower bits which are currently unused.
- enum REG_DIR_ENUM { REG_DIR_NONE = 0x00, REG_DIR_READ = 0x01, REG_DIR_WRITE = 0x02, REG_DIR_READ_WRITE = 0x03 }
- enum IDI_REG_ENUM
- enum SPI_CSB_ENUM { IDI_CSB_SOFTWARE = 0, IDI_CSB_BUFFER = 1, IDI_CSB_UINT8 = 2, IDI_CSB_UINT16 = 3 }
- enum { SPI_FIFO_SIZE = 16 }
- enum { HEX_DUMP_BYTES_PER_LINE = 16 }

  Dumps a hexadecimal and ASCII equivalent string to the desired output. The format, illustrated below is a classic memory dump format.

- enum { FRAM_DENSITY_BYTES = 8192 }

  FRAM Density current in use.

Functions

- static const char * IDI_Symbol_Name_Bank (IDI_BANK_ENUM bank)

  Outputs a human readable CSV to the desired output file or stdout.
- int IDI_Register_Report_CSV (const struct reg_definition *table, FILE *out)

  Outputs a human readable CSV to the desired output file or stdout.
- BOOL String_To_Bool (const char *str)

  General function used to convert a string into a boolean equivalent value.
- int Hex_Dump_Line (uint16_t address, size_t count, uint8_t *buffer, FILE *out)
- const char * EC_Code_To_Human_Readable (EC_ENUM error_code)
- BOOL Character_Get (int *character)

  Obtains a key from the keyboard in a non-blocking way. This is exerpetted from AES Universal Library/Driver.
- static char * IO_Get_Symbol_Name (IDI_REG_ENUM location)

  Translates a register enumerated symbol into a string that is the same as the enumerated symbol used throughout this code base.
- static BOOL IO_Direction_IsNotValid (IDI_REG_ENUM location, REG_DIR_ENUM direction)

  Looks up in the register definitions list for the ports possible read/write directions.
- int IO_Write_U8 (IDI_REG_ENUM location, uint8_t value)

  Writes uint8_t to I/O port. Macros are used to guide the target implementation.
- int IO_Read_U8 (IDI_REG_ENUM location, uint8_t *value)

  Reads uint8_t from I/O port. Macros are used to guide the target implementation.
- void IO_Write_U16_Address_Increment (IDI_REG_ENUM location, uint16_t value)

  Writes uint16_t to I/O ports in a uint8_t succession incrementing the offset address. Macros are used to guide the target implementation. In this case, bus width (which we typcially refer to the port width, which is different than register width) is assumed to be byte (uint8_t) wide.
- void IO_Read_U16_Address_Increment (IDI_REG_ENUM location, uint16_t *value)

  Reads uint16_t from I/O ports in a uint8_t succession incrementing the offset address. Macros are used to guide the target implementation. In this case, bus width (which we typcially refer to the port width, which is different than register width) is assumed to be byte (uint8_t) wide.
- void IO_Write_U16_Address_Fixed (IDI_REG_ENUM location, uint16_t value)

  Writes uint16_t to I/O ports in a uint8_t succession to the same address location. Macros are used to guide the target implementation. In this case, bus width (which we typcially refer to the port width, which is different than register width) is assumed to be byte (uint8_t) wide.
- void IO_Read_U16_Address_Fixed (IDI_REG_ENUM location, uint16_t *value)

  Reads uint16_t from I/O ports in a uint8_t succession to the same address location. Macros are used to guide the target implementation. In this case, bus width (which we typcially refer to the port width, which is different than register width) is assumed to be byte (uint8_t) wide.
- int IDI_DIN_ID_Get (uint16_t *id)

  Obtains the DIN component (or board ID in this case) ID number.
- BOOL IDI_DIN_IsNotPresent (void)

  Determines if the DIN component and/or board is present. Returns true if not present (i.e. error).
- static int IDI_DIN_Channel_Get (size_t channel, BOOL *value)

  Obtains and reports a single digital input channel.
- static int IDI_DIN_Group_Get (size_t group, uint8_t *value)

  Reads the selected digital input port (8-bits).
- int SPI_ID_Get (uint16_t *id)

Retrieves the SPI ID register value.

- int SPI_IsNotPresent (void)

  Reports if the SPI component is available within the register space by matching a known ID. The SPI register map is only enabled within the hardware if the hardware mode is not zero (i.e. M1 and M0 jumpers on the board provide a nonzero value).

- int SPI_Calculate_Half_Clock (double half_clock_request_sec, double *half_clock_actual_sec, double *error, uint16_t *hci)

  Computes the half clock register value given a requested time interval. It will also produce a 'report' indicating the actual value (due to integer resolution) as well as a computed error between requested and actual. The error can be used to determine whether timing constraints are met.

- int SPI_Calculate_Clock (double clock_request_hz, double *clock_actual_hz, double *error, uint16_t *hci)

  Computes the SPI clock half clock register value given a requested SPI clock frequency. It will also produce a 'report' indicating the actual value (due to integer resolution) as well as a computed error between requested and actual. The eror can be used to determine whether timing constraints are met.

- double SPI_Calculate_Half_Clock_Interval_Sec (uint16_t half_clock_interval)

  Computes the half clock interval in seconds given the value from the half clock interval register.

- int SPI_Calculate_End_Cycle_Delay (double spi_half_clock_interval_sec, double delay_request_sec, double *delay_actual_sec, double *error, uint8_t *ecd)

  Computes the time delay at the end of each byte transmitted. It will only output the parameters whose pointers are not NULL.

- int SPI_Configuration_Chip_Select_Behavior_Get (SPI_CSB_ENUM *chip_select_behavior)

  Extracts the chip select behavior from the SPI configuration register.

- int SPI_Configuration_Chip_Select_Behavior_Set (SPI_CSB_ENUM chip_select_behavior)

  Sets the chip select behavior to the SPI configuration register.

- int SPI_Configuration_Set (struct spi_cfg *cfg)

  Commits the configuration data structure to the hardware.

- int SPI_Configuration_Get (struct spi_cfg *cfg)

  Obtains the SPI configuration from the hardware.

- int SPI_Report_Configuration_Text (struct spi_cfg *cfg, FILE *out)

  Creates a human readable report of the SPI configuration data structure.

- int SPI_Report_Status_Text (struct spi_status *status, FILE *out)

  Produces a human readable report of the SPI status data structure.

- int SPI_Configuration_Initialize (struct spi_cfg *cfg)

  Initializes the SPI configuration data structure.

- int SPI_Status_Write (struct spi_status *status)

  Builds a detailed status data structure of the transmit/write outgoing SPI data FIFO. Reports the quantity of bytes currently in the transmit FIFO, full flag, empty flag, the total size of the FIFO in bytes, and sets tx_status to true indicating that this is status specific to the transmit FIFO.

- void SPI_Status_Write_FIFO_Status (BOOL *full, BOOL *empty, size_t *bytes_in_fifo)

  Returns the complete write/transmit FIFO status.

- BOOL SPI_Status_Write_FIFO_Is_Full (void)

  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.

- void SPI_Status_Read_FIFO_Status (BOOL *empty, size_t *bytes_available)

  Returns the complete read/receive FIFO status.

- int SPI_Status_Read (struct spi_status *status)

  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.

- BOOL SPI_Status_Write_FIFO_Is_Not_Empty (void)

Returns the transmit/write FIFO empty status flag. This function is typically used to wait for the transmit/write FIFO to become empty.

- BOOL SPI_Status_Read_FIFO_Is_Not_Empty (void)

  Returns the receive/read FIFO empty status flag. This function is typically used to determine if the FIFO is empty.

- int SPI_Commit (uint8_t chip_select)

  Sets/Clears the chip select or used to commit the transmit/write FIFO to the spi interface. The mode of operation is dependent on the chip_select_behavior.

- int SPI_FIFO_Write (const void *buffer, size_t size, size_t count, FILE *fd_log)

  Writes specifically to the SPI transmit/write data FIFO. It does not attempt to correlate the number of transmit bytes with receive bytes. Its purpose is more for low level hardware testing. Note that this function has a signature identical to the fwrite() function (i.e. make use of function pointers to guide destination of data).

- int SPI_FIFO_Read (const void *buffer, size_t size, size_t count, FILE *fd_log)

  Reads from the SPI receive/read data FIFO. It does not attempt to correlate the number of transmit bytes with receive bytes. Its purpose is more for low level hardware testing. Note that this function has a signature identical to the fread() function (i.e. make use of function pointers to guide sourcing of data).

- static int SPI_Data_Write_Read_Helper (size_t size, size_t tx_count, const void *tx_buffer, size_t rx_size, const void *rx_buffer, BOOL active_tx, BOOL active_rx, SPI_CSB_ENUM csb)
- int SPI_Data_Write_Read (size_t size, size_t tx_count, const void *tx_buffer, size_t rx_size, const void *rx_buffer)

  This function will write/read virtually any kind of data with almost any kind of chips select wrapping surrounding the data.

- int SPI_Data_Write (const void *tx_buffer, size_t size, size_t tx_count, FILE *fd_log)

  Special case of Write/Read that has a function signature same as fread() or fwrite().

- int SPI_Data_Read (const void *rx_buffer, size_t size, size_t rx_size, FILE *fd_log)

  Special case of Write/Read that has a function signature same as fread() or fwrite().

- int FRAM__Write_Enable_Latch_Set (void)

  FRAM Write Enable Latch Set command (WREN)

- int FRAM__Write_Disable (void)

  FRAM Write Latch disable (or clear) command (WRDI).

- int FRAM__Read_Status_Register (uint8_t *status)

  Read the FRAM status register and output the value.

- int FRAM__Write_Status_Register (uint8_t status)

  Write to the FRAM status register.

- int FRAM__Memory_Read (uint16_t address, size_t count, uint8_t *buffer)

  Reads data from FRAM memory to the output buffer.

- int FRAM__Memory_Write (uint16_t address, size_t count, uint8_t *buffer)

  Writes data from buffer to FRAM memory.

- int FRAM__Read_ID (uint32_t *id)
- int FRAM_Set (size_t count, uint8_t *buffer)

  This function will be used when creating a memory pool so that as blocks are allocated one can determine if we have an issue outside of any allocated space (i.e. overflows and so on).

- int FRAM_Report (uint16_t address, size_t length, FILE *out)
- int FRAM_Memory_To_File (uint16_t address, size_t length, FILE *binary)
- int FRAM_File_To_Memory (uint16_t address, size_t length, FILE *binary)
- static int IDI_CMD__SPI_ID (int argc, char *argv[])
- static int IDI_CMD__SPI_Config_Get (int argc, char *argv[])
- static int IDI_CMD__SPI_Config_Clock_Hz (int argc, char *argv[])
- static int IDI_CMD__SPI_Config_End_Cycle_Delay_Sec (int argc, char *argv[])
- static int IDI_CMD__SPI_Config_Mode (int argc, char *argv[])

  CPOL CPHA MODE 0 0 0 1 0 1 2 1 0 3 1 1.

- static int IDI_CMD__SPI_Config_SDI_Polarity (int argc, char *argv[])

- static int IDI_CMD__SPI_Config_SDO_Polarity (int argc, char *argv[])
- static int IDI_CMD__SPI_Config_SDIO_Wrap (int argc, char *argv[])
- static int IDI_CMD__SPI_Config_Chip_Select_Behavior (int argc, char *argv[])
- static int IDI_CMD__SPI_Status (int argc, char *argv[])
- static int IDI_CMD__SPI_Data (int argc, char *argv[])
- static int IDI_CMD__SPI_FIFO (int argc, char *argv[])
- static int IDI_CMD__SPI_Commit (int argc, char *argv[])
- int IDI_Command_Line_SPI (int argc, char *argv[])
- static int IDI_CMD__FRAM_Dump (int argc, char *argv[])
- static int IDI_CMD__FRAM_Save (int argc, char *argv[])
- static int IDI_CMD__FRAM_Load (int argc, char *argv[])
- static int IDI_CMD__FRAM_Init (int argc, char *argv[])
- static int IDI_CMD__FRAM_WREN (int argc, char *argv[])
- static int IDI_CMD__FRAM_WRDI (int argc, char *argv[])
- static int IDI_CMD__FRAM_RDSR (int argc, char *argv[])
- static int IDI_CMD__FRAM_WRSR (int argc, char *argv[])
- static int IDI_CMD__FRAM_RDID (int argc, char *argv[])
- int IDI_Command_Line_FRAM (int argc, char *argv[])
- static int IDI_CMD__DIN_ID (int argc, char *argv[])
- static int IDI_CMD__DIN_All (int argc, char *argv[])
- static int IDI_CMD__DIN_Channel (int argc, char *argv[])
- static int IDI_CMD__DIN_Group (int argc, char *argv[])
- int IDI_Command_Line_Digital_Input (int argc, char *argv[])
- int IDI_Command_Line_Register_Transaction (int argc, char *argv[])

  Either reads or writes a register using the form: idi <register acronym>="> [ ].
- static int IDI_CMD__Main_IO_Behavior (int argc, char *argv[])
- static int IDI_CMD__Main_Base (int argc, char *argv[])
- int IDI_Command_Line_Set (int argc, char *argv[])
- int IDI_Command_Line_Main (int argc, char *argv[])

  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.
- void IDI_Help (FILE *out)

  Outputs a help listing to the user.
- int IDI_Termination (void)

  Runs upon application exit. It saves the idi_dataset data structure.
- int IDI_Initialization (void)

  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.
- int main (int argc, char *argv[])

  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.

Variables

- static const char idi_svn_revision_string [] = { IDI_REV }

  Global variables.
- const struct ec_human_readable ec_human_readable []
- static const char * idi_bank_symbol_names []
- static const struct reg_definition definitions []

- struct idi_dataset idi_dataset
- static const char ec_unknown [] = "unknown error code"

  Translates an error code into a human readable message.
- static struct command_line idi_cmd_spi []
- static struct command_line idi_cmd_fram []
- static struct command_line idi_cmd_din []
- static struct command_line idi_cmd_main []

4.1.1 Macro Definition Documentation

4.1.1.1 #define CLOCK_PERIOD_SEC 20.0e-9

Board clock period as defined by the on-board oscillator which is 50MHz.

Definition at line 81 of file idi.c.

Referenced by SPI_Calculate_End_Cycle_Delay(), SPI_Calculate_Half_Clock(), SPI_Calculate_Half_Clock_Interval↩_Sec(), and SPI_Configuration_Get().

4.1.1.2 #define EC_EXTRACT_ENUM(*symbol, code, message*) symbol = code,

Definition at line 160 of file idi.c.

4.1.1.3 #define EC_EXTRACT_HUMAN_READABLE(*symbol, code, message*) { code, message },

Definition at line 161 of file idi.c.

4.1.1.4 #define EC_HUMAN_READABLE_TERMINATE { 0, NULL }

Definition at line 162 of file idi.c.

4.1.1.5 #define false 0

Definition at line 72 of file idi.c.

4.1.1.6 #define FRAM_BLOCK_SIZE 256

Definition at line 372 of file idi.c.

Referenced by FRAM_File_To_Memory(), FRAM_Memory_To_File(), and FRAM_Set().

4.1.1.7 #define ID_ALWAYS_REPORT_AS_GOOD 1

Definition at line 124 of file idi.c.

4.1.1.8 #define IDI_DIN_GROUP_QTY 6

Definition at line 355 of file idi.c.

Referenced by IDI_CMD__DIN_All(), and IDI_CMD__DIN_Group().

4.1.1.9 #define IDI_DIN_GROUP_SIZE 8

Definition at line 353 of file idi.c.

Referenced by IDI_DIN_Channel_Get().

4.1.1.10 #define IDI_DIN_QTY (IDI_DIN_GROUP_SIZE * IDI_DIN_GROUP_QTY)

Definition at line 356 of file idi.c.

4.1.1.11 #define IDI_DIN_SHIFT_RIGHT 3

Definition at line 354 of file idi.c.

Referenced by IDI_DIN_Channel_Get().

4.1.1.12 #define IDI_ERROR_CODES(_)

Value:

```
/*    enum_symbol                      code     human readable            */ \
        _( SUCCESS,                         0,      ""                          ) \
        _( EC_BUFFER_TOO_LARGE,             1,      "buffer too large"          ) \
        _( EC_DIRECTION,                    2,      "direction"                 ) \
        _( EC_PARAMETER,                    3,      "parameter"                 ) \
        _( EC_NOT_FOUND,                    4,      "not found"                 ) \
        _( EC_PARAMETER_MISSING,            5,      "missing parameter"         ) \
        _( EC_SYNTAX,                       6,      "syntax error"              ) \
        _( EC_HEX_DUMP_COUNT,               7,      "hex dump count"            ) \
        _( EC_INIT_FILE,                    8,      "write init file failed"    ) \
        _( EC_SPI_ECD_OUT_OF_RANGE,        20,      "SPI ECD range"             ) \
        _( EC_SPI_HALF_CLOCK_OUT_OF_RANGE, 21,      "SPI half clock range"      ) \
        _( EC_SPI_CSB_OUT_OF_RANGE,        22,      "SPI CSB range"             ) \
        _( EC_SPI_NOT_FOUND,               23,      "SPI not found"             ) \
        _( EC_SPI_BUFFER_SIZE_ODD,         24,      "buffer size odd"           ) \
        _( EC_SPI_BUFFER_SIZE,             25,      "buffer size out of range"  ) \
        _( EC_SPI_OBJECT_SIZE,             26,      "spi tx/rx object size"     )
```

An organized error code listing. This macro is used to build the complete error code enumeration list.

Definition at line 139 of file idi.c.

4.1.1.13 #define IDI_IO_DIRECTION_TEST 1

Definition at line 680 of file idi.c.

4.1.1.14 #define IDI_MESSAGE_SIZE 256

Definition at line 373 of file idi.c.

4.1.1.15 #define IDI_REGISTER_SET_DEFINITION(_)

Organized list of registers and associate attributes of each of the registers. This macro does not consume any memory of in itself, and is only 'consumed' or used to automatically build enumerations and pre-built data structures.

Definition at line 222 of file idi.c.

4.1.1.16 #define IDI_REV "$Date: 2015-03-05 08:48:57 -0600 (Thu, 05 Mar 2015) $"

The Subversion (SVN) time/date marker which is updated during commit of the source file to the repository.

Definition at line 48 of file idi.c.

4.1.1.17 #define REG_BANK_SET(*bank*) (bank << 8)

Definition at line 285 of file idi.c.

4.1.1.18 #define REG_EXTRACT_DEFINITION(*symbol, index, offset, register_bytes, aperture_bytes, read_write, name, bank*) { ((IDI_REG_ENUM) (REG_BANK_SET(bank) | REG_OFFSET_SET(offset))), read_write, bank, offset, #symbol, name },

Definition at line 293 of file idi.c.

4.1.1.19 #define REG_EXTRACT_ENUM(*symbol, index, offset, register_bytes, aperture_bytes, read_write, name, bank*) symbol = (REG_BANK_SET(bank) | REG_OFFSET_SET(offset)),

Definition at line 291 of file idi.c.

4.1.1.20 #define REG_LOCATION_BANK_GET(*location*) ((location >> 8) & 0xFF)

Definition at line 287 of file idi.c.

Referenced by IO_Read_U8(), and IO_Write_U8().

4.1.1.21 #define REG_LOCATION_OFFSET_GET(*location*) (location & 0xff)

Definition at line 288 of file idi.c.

Referenced by IO_Read_U8(), and IO_Write_U8().

4.1.1.22 #define REG_OFFSET_SET(*offset*) (offset & 0xff)

Definition at line 286 of file idi.c.

4.1.1.23 #define SPI_BLOCK_SIZE 256

Definition at line 374 of file idi.c.

4.1.1.24 #define strcmpi strcasecmp

Definition at line 36 of file idi.c.

Referenced by IDI_CMD__DIN_All(), IDI_CMD__DIN_Group(), IDI_CMD__Main_IO_Behavior(), IDI_CMD__SPI_Config_Chip_Select_Behavior(), IDI_CMD__SPI_FIFO(), IDI_CMD__SPI_Status(), IDI_Command_Line_Digital_Input(), IDI_Command_Line_FRAM(), IDI_Command_Line_Main(), IDI_Command_Line_Register_Transaction(), IDI_Command_Line_Set(), IDI_Command_Line_SPI(), and main().

4.1.1.25 #define true 1

Definition at line 73 of file idi.c.

4.1.2 Typedef Documentation

4.1.2.1 typedef int BOOL

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.

Boolean logic definitions

Definition at line 71 of file idi.c.

4.1.3 Enumeration Type Documentation

4.1.3.1 anonymous enum

Enumerator

FALSE

TRUE

Definition at line 74 of file idi.c.

```
74 { FALSE = 0, TRUE = 1 };
```

4.1.3.2 anonymous enum

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 Win↩ Systems 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 register space.

One could argue that there ought to be a nibble for revision and an upper nibble for the component ID itself, but quickly you find out the limitation of that implementation. If we simply use an arbitrary list from 1 to 65535 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 likely move to 32-bit and simply continue where we left off. The software really does not care, so long as the numbers are always unique. We can use the uniqueness of the IDs along with some implied intelligence (i.e. what board is this, what bus are we on, and perhaps a little bit of knowledge of the component itself) we can always build and grow software to accommodate new revisions and still support older revision hardware.

Enumerator

ID_DIN component DIN48 ID

ID_SPI component SPI ID

Definition at line 114 of file idi.c.

```
115 {
116     ID_DIN = 0x8012,
117     ID_SPI = 0x8013
118 };
```

4.1.3.3 anonymous enum

Enumerator

SPI_FIFO_SIZE

Definition at line 324 of file idi.c.

```
324 { SPI_FIFO_SIZE = 16 };
```

4.1.3.4 anonymous enum

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>=""">

Parameters

in	*address*	starting address.
in	*count*	number of bytes in the buffer
in	*buffer*	buffer containing the bytes to output
out	*out*	desired output destination.

Returns

An error code is returned.

Enumerator

HEX_DUMP_BYTES_PER_LINE

Definition at line 570 of file idi.c.

```
570 { HEX_DUMP_BYTES_PER_LINE = 16 };
```

4.1.3.5 anonymous enum

FRAM Density current in use.

Enumerator

FRAM_DENSITY_BYTES

Definition at line 2059 of file idi.c.

```
2059 { FRAM_DENSITY_BYTES = 8192 };
```

4.1.3.6 enum EC_ENUM

Definition at line 167 of file idi.c.

```
168 {
169     IDI_ERROR_CODES( EC_EXTRACT_ENUM )
170 } EC_ENUM;  /* EC = Error Code */
```

4.1.3.7 enum IDI_BANK_ENUM

The bank register mapping. This mapping is upwardly compatible with the legacy hardware banking register. It also allows for future expansion utilizing the lower bits which are currently unused.

Enumerator

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 indicates exclusive of bank address

IDI_BANK_UNDEFINED indicates that no banking has been defined

Definition at line 191 of file idi.c.

```
192 {
193     IDI_BANK_0            = 0x00,
194     IDI_BANK_1            = 0x40,
195     IDI_BANK_2            = 0x80,
196     IDI_BANK_3            = 0xC0,
197     IDI_BANK_4            = 0x20,
198     IDI_BANK_5            = 0x60,
199     IDI_BANK_6            = 0xA0,
200     IDI_BANK_7            = 0xE0,
201     IDI_BANK_NONE      = 0xFE,
202     IDI_BANK_UNDEFINED  = 0xFF
203 } IDI_BANK_ENUM;
```

4.1.3.8 enum IDI_REG_ENUM

Definition at line 296 of file idi.c.

```
297 {
298     IDI_REGISTER_SET_DEFINITION( REG_EXTRACT_ENUM )
299 } IDI_REG_ENUM;
```

4.1.3.9 enum REG_DIR_ENUM

Enumerator

REG_DIR_NONE

REG_DIR_READ

REG_DIR_WRITE

REG_DIR_READ_WRITE

Definition at line 205 of file idi.c.

```
206 {
207     REG_DIR_NONE         = 0x00,
208     REG_DIR_READ         = 0x01,
209     REG_DIR_WRITE      = 0x02,
210     REG_DIR_READ_WRITE    = 0x03
211 } REG_DIR_ENUM;
```

4.1.3.10 enum SPI_CSB_ENUM

Enumerator

IDI_CSB_SOFTWARE

IDI_CSB_BUFFER

IDI_CSB_UINT8

IDI_CSB_UINT16

Definition at line 315 of file idi.c.

```
316 {
317     IDI_CSB_SOFTWARE    = 0,
318     IDI_CSB_BUFFER        = 1,
319     IDI_CSB_UINT8      = 2,
320     IDI_CSB_UINT16        = 3
321 } SPI_CSB_ENUM;
```

4.1.4 Function Documentation

4.1.4.1 BOOL Character_Get (int * *character*)

Obtains a key from the keyboard in a non-blocking way. This is exerpetted from AES Universal Library/Driver.

Parameters

out	*character*	Character from keyboard otherwise null character.

Returns

If true, then a valid character is available at the keyboard, otherwise false.

Definition at line 652 of file idi.c.

Referenced by main().

```
653 {
654     int  char_temp;
655     BOOL result = false;
656 #ifdef __BORLANDC__

657     if ( kbhit() )
658     {
659         char_temp = getch();
660         result    = true;
661     }
662     else
663     {
664         char_temp = '\0';
665     }
666 #else

667     char_temp = '\0';
668     result    = false;
669 #endif

670     if ( NULL != character ) *character = char_temp;
671     return result;
672 }
```

4.1.4.2 const char* EC_Code_To_Human_Readable (EC_ENUM *error_code*)

Definition at line 626 of file idi.c.

References ec_unknown, ec_human_readable::message, and idi_dataset::message.

Referenced by main().

```
627 {
628     int index;
629
630     if ( error_code < 0 ) error_code = -error_code;
631
632     index = 0;
633     while( NULL != ec_human_readable[index].message )
634     {
635         if ( ((EC_ENUM) error_code) == ec_human_readable[index].error_code )
636         {
637             return ec_human_readable[index].message;
638         }
639         index++;
640     }
641     return ec_unknown;
642 }
```

4.1.4.3 int FRAM__Memory_Read (uint16_t *address*, size_t *count*, uint8_t * *buffer*)

Reads data from FRAM memory to the output buffer.

Parameters

in	*address*	Starting FRAM address
in	*count*	Number of bytes to transfer
out	*buffer*	Destination buffer in which to store the data

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2142 of file idi.c.

References IDI_CSB_SOFTWARE, SPI_Commit(), SPI_Configuration_Chip_Select_Behavior_Set(), SPI_Data_↩ Write_Read(), and SPI_Status_Write_FIFO_Is_Not_Empty().

Referenced by FRAM_Memory_To_File(), and FRAM_Report().

```
2143 {
2144     int             error_code;
2145 //  SPI_CSB_ENUM    csb_copy;
2146     uint8_t         tx_buf[3] = { 0x03, 0x00, 0x00 }; /* opcode: READ = 0x03 */
2147     //uint8_t rx_buf[3];
2148
2149     tx_buf[1] = (uint8_t)( address & 0xFF );
2150     tx_buf[2] = (uint8_t)( address >>   8 );
2151
2152 //  /* retain an existing copy of the actual csb value */
2153 //  error_code = SPI_Configuration_Chip_Select_Behavior_Get( &csb_copy );
2154 //  if ( error_code ) return error_code;
2155     /* over-ride and set it to what we wish it to be */
2156     error_code = SPI_Configuration_Chip_Select_Behavior_Set(
      IDI_CSB_SOFTWARE );
2157     if ( error_code ) return error_code;
2158
2159     SPI_Commit( 1 );
2160
2161     error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 3, tx_buf, 0, NULL );
2162     if ( error_code ) return error_code;
2163
2164     error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 0, NULL, count, buffer );
2165     if ( error_code ) return error_code;
2166
2167     while ( SPI_Status_Write_FIFO_Is_Not_Empty() ); /* wait for buffer to
       empty */
2168     SPI_Commit( 0 );
2169
2170 //  /* restore the csb */
2171 //  error_code = SPI_Configuration_Chip_Select_Behavior_Set( csb_copy );
2172 //  if ( error_code ) return error_code;
2173
2174     return SUCCESS;
2175 }
```

4.1.4.4 int FRAM__Memory_Write (uint16_t *address*, size_t *count*, uint8_t * *buffer*)

Writes data from buffer to FRAM memory.

Parameters

in	*address*	Starting FRAM address
in	*count*	Number of bytes to transfer
out	*buffer*	Source buffer from which data will be transfered to FRAM

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2187 of file idi.c.

References IDI_CSB_SOFTWARE, SPI_Commit(), SPI_Configuration_Chip_Select_Behavior_Set(), SPI_Data_↩ Write_Read(), and SPI_Status_Write_FIFO_Is_Not_Empty().

Referenced by FRAM_File_To_Memory(), and FRAM_Set().

```
2188 {
2189     int             error_code;
2190 //  SPI_CSB_ENUM    csb_copy;
2191     uint8_t         tx_buf[3] = { 0x02, 0x00, 0x00 }; /* opcode: WRITE = 0x02 */
2192
2193     tx_buf[1] = (uint8_t)( address & 0xFF );
```

```
2194     tx_buf[2] = (uint8_t)( address >>   8 );
2195
2196 //  /* retain an existing copy of the actual csb value */
2197 //  error_code = SPI_Configuration_Chip_Select_Behavior_Get( &csb_copy );
2198 //  if ( error_code ) return error_code;
2199     /* over-ride and set it to what we wish it to be */
2200     error_code = SPI_Configuration_Chip_Select_Behavior_Set(
     IDI_CSB_SOFTWARE );
2201     if ( error_code ) return error_code;
2202
2203     SPI_Commit( 1 );
2204
2205     error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 3, tx_buf, 0, NULL );
2206     if ( error_code ) return error_code;
2207
2208     error_code = SPI_Data_Write_Read( sizeof( uint8_t ), count, buffer, 0, NULL );
2209     if ( error_code ) return error_code;
2210
2211     while ( SPI_Status_Write_FIFO_Is_Not_Empty() ); /* wait for buffer to
      empty */
2212     SPI_Commit( 0 );
2213
2214 //  /* restore the csb */
2215 //  error_code = SPI_Configuration_Chip_Select_Behavior_Set( csb_copy );
2216 //  if ( error_code ) return error_code;
2217     return SUCCESS;
2218 }
```

4.1.4.5 int FRAM__Read_ID (uint32_t * *id*)

Parameters

out	*id*	The 32-bit ID register read from the FRAM. This appears to be only available with Fujistu parts.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2228 of file idi.c.

References IDI_CSB_BUFFER, SPI_Configuration_Chip_Select_Behavior_Set(), and SPI_Data_Write_Read().

Referenced by FRAM_Set(), and IDI_CMD__FRAM_RDID().

```
2229 {
2230     int         error_code;
2231     uint8_t     tx_buf[5] = { 0x9F, 0x00, 0x00, 0x00, 0x00 }; /* opcode: RDID = 0x9F */
2232     uint8_t     rx_buf[5];
2233
2234     SPI_Configuration_Chip_Select_Behavior_Set(
     IDI_CSB_BUFFER );
2235     error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 5, tx_buf, 5, rx_buf );
2236     if ( error_code ) return error_code;
2237
2238     {
2239         int id_scratch = 0;
2240         int index;
2241
2242         for ( index = 4; index > 0; index-- )
2243         {
2244             id_scratch = ( id_scratch << 8 ) | ( (uint32_t) rx_buf[index] );
2245         }
2246         *id = id_scratch;
2247     }
2248
2249     return SUCCESS;
2250 }
```

4.1.4.6 int FRAM__Read_Status_Register (uint8_t * *status*)

Read the FRAM status register and output the value.

Parameters

status	A pointer to the destination location of the status data.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2098 of file idi.c.

References IDI_CSB_BUFFER, SPI_Configuration_Chip_Select_Behavior_Set(), and SPI_Data_Write_Read().

Referenced by IDI_CMD__FRAM_RDSR().

```
2099 {
2100     int     error_code;
2101     uint8_t tx_buf[2] = { 0x05, 0x00 }; /* opcode: RDSR = 0x04 */
2102     uint8_t rx_buf[2];
2103
2104     SPI_Configuration_Chip_Select_Behavior_Set(
      IDI_CSB_BUFFER );
2105     error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 2, tx_buf, 2, rx_buf );
2106     if ( error_code ) return error_code;
2107     *status = rx_buf[1];
2108     return SUCCESS;
2109 }
```

4.1.4.7 int FRAM__Write_Disable (void)

FRAM Write Latch disable (or clear) command (WRDI).

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2083 of file idi.c.

References IDI_CSB_BUFFER, SPI_Configuration_Chip_Select_Behavior_Set(), and SPI_Data_Write_Read().

Referenced by IDI_CMD__FRAM_WRDI().

```
2084 {
2085     uint8_t tx_buf[1] = { 0x04 }; /* opcode: WRDI = 0x04 */
2086     SPI_Configuration_Chip_Select_Behavior_Set(
      IDI_CSB_BUFFER );
2087     return SPI_Data_Write_Read( sizeof( uint8_t ), 1, tx_buf, 0, NULL );
2088 }
```

4.1.4.8 int FRAM__Write_Enable_Latch_Set (void)

FRAM Write Enable Latch Set command (WREN)

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2069 of file idi.c.

References IDI_CSB_BUFFER, SPI_Configuration_Chip_Select_Behavior_Set(), and SPI_Data_Write_Read().

Referenced by IDI_CMD__FRAM_WREN().

```
2070 {
2071     uint8_t tx_buf[1] = { 0x06 }; /* opcode: WREN = 0x06 */
2072     SPI_Configuration_Chip_Select_Behavior_Set(
     IDI_CSB_BUFFER );
2073     return SPI_Data_Write_Read( sizeof( uint8_t ), 1, tx_buf, 0, NULL );
2074 }
```

4.1.4.9 int FRAM__Write_Status_Register (uint8_t *status*)

Write to the FRAM status register.

Parameters

in	*FRAM*	status value to be written.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2119 of file idi.c.

References IDI_CSB_BUFFER, SPI_Configuration_Chip_Select_Behavior_Set(), and SPI_Data_Write_Read().

Referenced by IDI_CMD__FRAM_WRSR().

```
2120 {
2121     int     error_code;
2122     uint8_t tx_buf[2] = { 0x05, 0x00 }; /* opcode: RDSR = 0x04 */
2123     uint8_t rx_buf[2];
2124
2125     tx_buf[1] = status;
2126     SPI_Configuration_Chip_Select_Behavior_Set(
     IDI_CSB_BUFFER );
2127     error_code = SPI_Data_Write_Read( sizeof( uint8_t ), 2, tx_buf, 2, rx_buf );
2128     if ( error_code ) return error_code;
2129     return SUCCESS;
2130 }
```

4.1.4.10 int FRAM_File_To_Memory (uint16_t *address*, size_t *length*, FILE * *binary*)

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2393 of file idi.c.

References FRAM__Memory_Write(), idi_dataset::fram_block, and FRAM_BLOCK_SIZE.

Referenced by IDI_CMD__FRAM_Load().

```
2394 {
2395     int     error_code;
2396     size_t  count_read;
2397     size_t  count_total;
2398     size_t  count_actual;
2399
2400     count_total = 0;
2401     count_read  = FRAM_BLOCK_SIZE;
2402     if ( 0 == length )
2403     {
2404         do
2405         {
2406             count_actual = fread( idi_dataset.fram_block, 1, count_read, binary );
2407             error_code   = FRAM__Memory_Write( address, count_read,
```

```
    idi_dataset.fram_block );
2408            if ( error_code ) return error_code;
2409            count_total += count_actual;
2410            if ( count_actual != count_read ) count_read = 0; /* must be at end of file */
2411        } while ( count_read > 0 );
2412    }
2413    else
2414    {
2415        if ( length < count_read ) count_read = length;
2416        do
2417        {
2418            count_actual = fread( idi_dataset.fram_block, 1, count_read, binary );
2419            error_code   = FRAM__Memory_Write( address, count_read,
    idi_dataset.fram_block );
2420            if ( error_code ) return error_code;
2421            count_total += count_actual;
2422            length      -= count_actual;
2423            if ( count_actual != count_read ) count_read = 0; /* must be at end of file */
2424            if ( length < count_read ) count_read = length;
2425        } while ( count_read > 0 );
2426    }
2427    return SUCCESS;
2428 }
```

4.1.4.11 int FRAM_Memory_To_File (uint16_t *address*, size_t *length*, FILE * *binary*)

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2362 of file idi.c.

References FRAM__Memory_Read(), idi_dataset::fram_block, and FRAM_BLOCK_SIZE.

Referenced by IDI_CMD__FRAM_Save().

```
2363 {
2364     int         error_code;
2365     size_t      block_count;
2366     size_t      block_remainder;
2367
2368     block_count = length / ((size_t) FRAM_BLOCK_SIZE);
2369     block_remainder = length - ( block_count * ((size_t) FRAM_BLOCK_SIZE) );
2370
2371     while ( block_count > 0 )
2372     {
2373         error_code = FRAM__Memory_Read( address, ((size_t)
    FRAM_BLOCK_SIZE), idi_dataset.fram_block );
2374         fwrite( idi_dataset.fram_block, 1, ((size_t)
    FRAM_BLOCK_SIZE), binary );
2375         address += FRAM_BLOCK_SIZE;
2376         block_count--;
2377     }
2378
2379     if ( block_remainder > 0 )
2380     {
2381         error_code = FRAM__Memory_Read( address, block_remainder,
    idi_dataset.fram_block );
2382         fwrite( idi_dataset.fram_block, 1, block_remainder, binary );
2383         if ( error_code ) return error_code;
2384     }
2385     return SUCCESS;
2386 }
```

4.1.4.12 int FRAM_Report (uint16_t *address*, size_t *length*, FILE * *out*)

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2329 of file idi.c.

References FRAM__Memory_Read(), idi_dataset::fram_block, HEX_DUMP_BYTES_PER_LINE, and Hex_Dump_↩ Line().

Referenced by IDI_CMD__FRAM_Dump().

```
2330 {
2331     int         error_code;
2332     const int   block_size = HEX_DUMP_BYTES_PER_LINE;
2333     size_t      block_count;
2334     size_t      block_remainder;
2335
2336     block_count = ((size_t) length) / block_size;
2337     block_remainder = ((size_t) length) - ( block_count * block_size );
2338
2339     while ( block_count > 0 )
2340     { /* output a line at a time */
2341         error_code = FRAM__Memory_Read( address, block_size,
      idi_dataset.fram_block );
2342         error_code = Hex_Dump_Line( address, block_size,
      idi_dataset.fram_block, out );
2343         address += block_size;
2344         block_count--;
2345     }
2346
2347     if ( block_remainder > 0 )
2348     { /* output any remaining portion */
2349         error_code = FRAM__Memory_Read( address, block_remainder,
      idi_dataset.fram_block );
2350         error_code = Hex_Dump_Line( address, block_remainder,
      idi_dataset.fram_block, out );
2351         if ( error_code ) return error_code;
2352     }
2353
2354     return SUCCESS;
2355 }
```

4.1.4.13 int FRAM_Set (size_t *count*, uint8_t * *buffer*)

This function will be used when creating a memory pool so that as blocks are allocated one can determine if we have an issue outside of any allocated space (i.e. overflows and so on).

Parameters

in	*cfg*	pass in the configuration to be written to hardware.

Returns

a nonzero if successful, else return zero.

Writes a repeating pattern to the entire FRAM memory array. the pattern is obtained from the buffer. If the buffer is NULL, then all zeros are written to the FRAM.

Parameters

in	*count*	Length in bytes of the pattern found within the buffer

in	*buffer*	input buffer containing the repeat pattern to be written to FRAM

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2273 of file idi.c.

References FRAM__Memory_Write(), FRAM__Read_ID(), idi_dataset::fram_block, FRAM_BLOCK_SIZE, and FRAM_DENSITY_BYTES.

Referenced by IDI_CMD__FRAM_Init().

```
2274 {
2275     int         error_code;
2276     int         block_count;
2277     int         block_remainder;
2278     uint16_t    address;
2279     uint32_t    id;
2280
2281     error_code = FRAM__Read_ID( &id );
2282     if ( error_code ) return error_code;
2283
2284     address = 0;
2285
2286     if ( count > 1 )
2287     { /* */
2288         block_count     = ((int) FRAM_DENSITY_BYTES) / ((int) count );
2289         block_remainder = ((int) FRAM_DENSITY_BYTES) - ( block_count * ((int)
      FRAM_BLOCK_SIZE) );
2290     }
2291     else
2292     { /* only one fill character, so we create a buffer of it to make things a bit faster */
2293         int         index;
2294         const int   block_size = FRAM_BLOCK_SIZE;
2295         /* prefill */
2296         if ( NULL == buffer )
2297         {
2298             for ( index = 0; index < block_size; index++ ) idi_dataset.
      fram_block[index]= 0;
2299         }
2300         else
2301         {
2302             for ( index = 0; index < block_size; index++ ) idi_dataset.
      fram_block[index]= buffer[0];
2303         }
2304         block_count     = ((int) FRAM_DENSITY_BYTES) / block_size;
2305         block_remainder = ((int) FRAM_DENSITY_BYTES) - ( block_count * block_size );
2306         buffer          = idi_dataset.fram_block;
2307     }
2308
2309     while ( block_count > 0 )
2310     {
2311         error_code = FRAM__Memory_Write( address, ((int) count), buffer );
2312         address += (uint16_t) count;
2313         block_count--;
2314     }
2315
2316     if ( block_remainder > 0 )
2317     {
2318         error_code = FRAM__Memory_Write( address, block_remainder, buffer );
2319         if ( error_code ) return error_code;
2320     }
2321     return SUCCESS;
2322 }
```

4.1.4.14 int Hex_Dump_Line (uint16_t *address*, size_t *count*, uint8_t * *buffer*, FILE * *out*)

Definition at line 572 of file idi.c.

References HEX_DUMP_BYTES_PER_LINE.

Referenced by FRAM_Report(), IDI_CMD__SPI_Data(), and IDI_CMD__SPI_FIFO().

```
573 {
574     size_t  index;
575     char    str_temp[8];
576     char    str_ascii[20];
577     char    str_hex_list[64];
578 
579     //if ( count > 16 ) return -EC_BUFFER_TOO_LARGE;
580     if ( count > HEX_DUMP_BYTES_PER_LINE ) return -EC_HEX_DUMP_COUNT;
581 
582     sprintf( str_hex_list, "%04X: ", ((int) address) );
583     strcpy( str_ascii,    "" );
584     for ( index = 0; index < count; index++ )
585     {   /* append/build hex list */
586         sprintf( str_temp, "%02X", buffer[index] );
587         strcat( str_hex_list, str_temp );
588         /* output spacer in the middle and end */
589         if ( 0x07 == ( index & 0x07 ) ) strcat( str_hex_list, "  " );
590         /* add a space after hex value and spacers */
591         strcat( str_hex_list, " " );
592         /* append/build ASCII list */
593         if ( ( buffer[index] < ' ' ) || ( buffer[index] > '~' ) )
594         { /* since these characters will not display replace them with a period */
595             strcat( str_ascii, "." );
596         }
597         else
598         { /* print the character as is */
599             sprintf( str_temp, "%c", buffer[index] );
600             strcat( str_ascii, str_temp );
601         }
602     }
603     /* compute any remaining filler required to properly align ASCII portion */
604     index = strlen( str_hex_list );
605     /* total = ( 6 ) 'address characters' + ( 16 * 3 ) 'hex characters' + ( 2 * 3 ) 'spacers'  = 60 */
606     count = 60 - index;
607     while ( count > 0 )
608     { /* add sufficient characters so that the ASCII portion is in the proper columns */
609         strcat( str_hex_list, " " );
610         count--;
611     }
612     /* output the results */
613     fprintf( out, "%s%s\n", str_hex_list, str_ascii );
614     return SUCCESS;
615 }
```

4.1.4.15 static int IDI_CMD__DIN_All (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3284 of file idi.c.

References IDI_DIN_Group_Get(), IDI_DIN_GROUP_QTY, and strcmpi.

```
3285 { /* idi din all [<binary/hex/group>]  */
3286     int     error_code;            /* used primarily for debug purposes */
3287     int     channel;
3288     //BOOL  value;
3289     int     cp;
3290     enum { MODE_NONE = 0, MODE_BINARY = 1, MODE_HEX = 2, MODE_ALL = 3 } mode_out;
3291     int     group;
3292     char    message[64];
3293     uint8_t din_grp[6];
3294     uint8_t mask;
3295     (void)  argc;
3296     (void)  argv;
3297 
3298     mode_out = MODE_ALL;
3299     if ( argc > 0 )
3300     {
3301         int index;
```

```
3302         mode_out = MODE_NONE;
3303         for ( index = 0; index < argc; index++ )
3304         {
3305             if      ( 0 == strcmpi( "binary", argv[index] ) ) mode_out |= MODE_BINARY;
3306             else if ( 0 == strcmpi( "group",  argv[index] ) ) mode_out |= MODE_HEX;
3307             else if ( 0 == strcmpi( "hex",    argv[index] ) ) mode_out |= MODE_HEX;
3308             else                                              mode_out |= MODE_ALL;
3309         }
3310     }
3311     /* build in binary format */
3312     cp    = 0;
3313     group = 0;
3314     for ( group = 0; group < IDI_DIN_GROUP_QTY; group++ )
3315     {
3316         error_code = IDI_DIN_Group_Get( group, &(din_grp[group]) );
3317         mask = 0x01;
3318         for ( channel = 0; channel < 8; channel++ )
3319         {
3320             message[cp++] = !!(din_grp[group] & mask) ? '1' : '0';
3321             mask = mask << 1;
3322         }
3323         message[cp++] = ' ';
3324     }
3325     message[cp] = '\0';
3326     if ( MODE_BINARY == ( mode_out & MODE_BINARY ) )
3327     {
3328         printf( "DIN: %s\n", message );
3329     }
3330     if ( MODE_HEX == ( mode_out & MODE_HEX ) )
3331     {
3332         printf( "DIN:" );
3333         for ( group = 0; group < IDI_DIN_GROUP_QTY; group++ ) printf( " %02X", din_grp[
     group] );
3334         printf( "\n" );
3335     }
3336
3337     return SUCCESS;
3338 }
```

4.1.4.16 static int IDI_CMD__DIN_Channel (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3345 of file idi.c.

References IDI_DIN_Channel_Get().

```
3346 {
3347     int     error_code;           /* used primarily for debug purposes */
3348     int     channel;
3349     char    message[8];
3350     BOOL    value;
3351
3352     if ( argc < 1 ) return -EC_NOT_FOUND;
3353
3354     channel = (int) strtol( argv[0], NULL, 0 );
3355     error_code = IDI_DIN_Channel_Get( channel, &value );
3356     message[0] = value ? '1' : '0';
3357     message[1] = '\0';
3358
3359     printf( "DIN%02d: %s\n", channel, message );
3360     return SUCCESS;
3361 }
```

4.1.4.17 static int IDI_CMD__DIN_Group (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3368 of file idi.c.

References IDI_DIN_Group_Get(), IDI_DIN_GROUP_QTY, and strcmpi.

```
3369 {
3370     int     error_code;
3371 //  int     index;
3372     int     group;
3373     int     group_count;
3374     BOOL    do_all;
3375     uint8_t din_grp[6];
3376
3377     if ( argc < 1 ) do_all = true;
3378     else            do_all = false;
3379
3380     if ( 0 == strcmpi( "all", argv[0] ) ) do_all = true;
3381
3382     if ( do_all )
3383     { /* all */
3384         for ( group = 0; group < IDI_DIN_GROUP_QTY; group ++ )
3385         {
3386             error_code = IDI_DIN_Group_Get( group, &(din_grp[group]) );
3387         }
3388         group_count = IDI_DIN_GROUP_QTY;
3389     }
3390     else
3391     {
3392         group = (int) strtol( argv[0], NULL, 0 );
3393         error_code = IDI_DIN_Group_Get( group, &(din_grp[0]) );
3394         group_count = 1;
3395     }
3396
3397     printf( "DIN_GROUP:" );
3398     for ( group = 0; group < group_count; group++ )
3399     {
3400         printf( " 0x%02X", ((int) din_grp[group]) );
3401     }
3402     printf( "\n" );
3403     return SUCCESS;
3404 }
```

4.1.4.18 static int IDI_CMD__DIN_ID (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3269 of file idi.c.

References IDI_DIN_ID_Get().

```
3270 { /*  idi spi id */
3271     uint16_t    id;
3272     (void)      argc;
3273     (void)      argv;
3274     IDI_DIN_ID_Get( &id );
3275     printf( "DIN ID: 0x%04X\n", id );
3276     return SUCCESS;
3277 }
```

4.1.4.19 static int IDI_CMD__FRAM_Dump (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3054 of file idi.c.

References FRAM_DENSITY_BYTES, FRAM_Report(), and HEX_DUMP_BYTES_PER_LINE.

```
3055 { /* idi fram dump <address> <length>  */
3056     uint16_t address;
3057     uint16_t length;
3058
3059     if ( argc < 1 ) return -EC_PARAMETER;
3060
3061     address = (uint16_t) strtol( argv[0], NULL, 0 );
3062
3063     if ( argc < 2 ) length = HEX_DUMP_BYTES_PER_LINE;
3064     else            length  = (uint16_t) strtol( argv[1], NULL, 0 );
3065
3066     if ( ( address + length ) > FRAM_DENSITY_BYTES ) length =
      FRAM_DENSITY_BYTES - address;
3067
3068     return FRAM_Report( address, length, stdout );
3069 }
```

4.1.4.20 static int IDI_CMD__FRAM_Init (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3119 of file idi.c.

References FRAM_Set().

```
3120 { /*  idi fram init <pattern list: 0x55 0x33 '3' '5' 'q' >  */
3121     int         error_code;
3122     int         index;
3123     uint8_t     buf[16];
3124     if ( argc < 1 )
3125     { /* initialize all zeros */
3126         error_code = FRAM_Set( 0, NULL );
3127     }
3128     else
3129     {
3130         if ( argc > 16 ) argc = 16;
3131         for ( index = 0; index < argc; index++ )
3132         {
3133             buf[index] = (uint8_t) strtol( argv[index], NULL, 0 );
3134         }
3135         error_code = FRAM_Set( argc, buf );
3136     }
3137     return error_code;
3138 }
```

4.1.4.21 static int IDI_CMD__FRAM_Load (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3098 of file idi.c.

References FRAM_File_To_Memory().

```
3099 { /* idi fram load <address> <source_file> */
3100     int         error_code;
3101     uint16_t    address;
3102 //  uint16_t    length;
3103     FILE *      out;
3104
3105     if ( argc < 2 ) return -EC_PARAMETER;
3106
3107     address = (uint16_t) strtol( argv[0], NULL, 0 );
3108     out = fopen( argv[2], "r" );
3109     error_code = FRAM_File_To_Memory( address, 0 /* no length specified */, out );
3110     fclose( out );
3111     return error_code;
3112 }
```

4.1.4.22 static int IDI_CMD__FRAM_RDID (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3200 of file idi.c.

References FRAM__Read_ID().

```
3201 { /*  idi fram RDID  */
3202     uint32_t    id;
3203     (void)      argc;
3204     (void)      argv;
3205     FRAM__Read_ID( &id );
3206     printf( "FRAM ID: 0x%08X\n", id );
3207     return SUCCESS;
3208 }
```

4.1.4.23 static int IDI_CMD__FRAM_RDSR (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3169 of file idi.c.

References FRAM__Read_Status_Register().

```
3170 { /* idi fram RDSR  */
3171     int         error_code;
3172     uint8_t     status;
3173     (void)      argc;
3174     (void)      argv;
3175     error_code = FRAM__Read_Status_Register( &status );
3176     printf( "FRAM STATUS: 0x%02X\n", ((int) status) );
3177     return error_code;
3178 }
```

4.1.4.24 static int IDI_CMD__FRAM_Save (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3076 of file idi.c.

References FRAM_Memory_To_File().

```
3077 { /* idi fram save <address> <length> <destination_file> */
3078     int         error_code;
3079     uint16_t    address;
3080     size_t      length;
3081     FILE *      out;
3082
3083     if ( argc < 3 ) return -EC_PARAMETER;
3084
3085     address = (uint16_t) strtol( argv[0], NULL, 0 );
3086     length  = (uint16_t) strtol( argv[1], NULL, 0 );
3087     out = fopen( argv[2], "w" );
3088     error_code = FRAM_Memory_To_File( address, length, out );
3089     fclose( out );
3090     return error_code;
3091 }
```

4.1.4.25 static int IDI_CMD__FRAM_WRDI (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3157 of file idi.c.

References FRAM__Write_Disable().

```
3158 { /*  idi fram WRDI  */
3159     (void)      argc;
3160     (void)      argv;
3161     return FRAM__Write_Disable();
3162 }
```

4.1.4.26 static int IDI_CMD__FRAM_WREN (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3145 of file idi.c.

References FRAM__Write_Enable_Latch_Set().

```
3146 { /* idi fram WREN */
3147     (void)      argc;
3148     (void)      argv;
3149     return FRAM__Write_Enable_Latch_Set();
3150 }
```

4.1.4.27 static int IDI_CMD__FRAM_WRSR (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3185 of file idi.c.

References FRAM__Write_Status_Register().

```
3186 { /*  idi fram WRSR <value>  */
3187 //  int     error_code;
3188     uint8_t status;
3189
3190     if ( argc < 1 ) return -EC_PARAMETER;
3191     status = (uint8_t) strtol( argv[0], NULL, 0 );
3192     return FRAM__Write_Status_Register( status );
3193 }
```

4.1.4.28 static int IDI_CMD__Main_Base (int *argc,* char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3583 of file idi.c.

References idi_dataset::base_address.

```
3584 { /*  idi spi ecd [<time sec>] */
3585     int      error_code;              /* used primarily for debug purposes */
3586
3587     if ( argc < 1 )
3588     { /* read */
3589         printf( "BASE_ADDRESS: 0x%04X\n", idi_dataset.base_address );
3590     }
3591     else
3592     { /* write */
3593         idi_dataset.base_address = (uint16_t) strtol( argv[0], NULL, 0 );
3594         printf( "OK\n" );
3595     }
3596     return SUCCESS;
3597 }
```

4.1.4.29 static int IDI_CMD__Main_IO_Behavior (int *argc,* char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3543 of file idi.c.

References idi_dataset::io_report, idi_dataset::io_simulate, strcmpi, and String_To_Bool().

```
3544 { /*  idi spi ecd [<time sec>] */
3545     int      error_code;              /* used primarily for debug purposes */
3546
3547     if ( argc < 1 )
3548     { /* read */
3549         printf( "IO Simulate = %s\n", idi_dataset.io_simulate ? "true" : "false" );
3550         printf( "IO Report   = %s\n", idi_dataset.io_report   ? "true" : "false" );
3551     }
3552     else if ( argc > 1 )
3553     { /* write */
3554         if       ( 0 == strcmpi( "simulate", argv[0] ) )
3555         {
3556             idi_dataset.io_simulate = String_To_Bool( argv[1] );
```

```
3557         }
3558         else if  ( 0 == strcmpi( "report", argv[0] ) )
3559         {
3560             idi_dataset.io_report = String_To_Bool( argv[1] );
3561         }
3562         printf( "OK\n" );
3563     }
3564     else
3565     { /* read individual */
3566         if       ( 0 == strcmpi( "simulate", argv[0] ) )
3567         {
3568             printf( "IO Simulate = %s\n", idi_dataset.io_simulate ? "true" : "false"
     );
3569         }
3570         else if  ( 0 == strcmpi( "report", argv[0] ) )
3571         {
3572             printf( "IO Report   = %s\n", idi_dataset.io_report   ? "true" : "false" );
3573         }
3574     }
3575     return SUCCESS;
3576 }
```

4.1.4.30 static int IDI_CMD__SPI_Commit (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2975 of file idi.c.

References SPI_Commit(), and String_To_Bool().

```
2976 { /*  idi spi commit [<true/1/false/0>] */
2977     int      error_code;
2978     uint8_t  chip_select;
2979
2980     if      ( argc < 1                  ) chip_select = 0x01;
2981     else if ( String_To_Bool( argv[0] ) ) chip_select = 0x01;
2982     else                                  chip_select = 0x00;
2983
2984     error_code = SPI_Commit( chip_select );
2985     if ( error_code ) return error_code;
2986
2987     printf( "OK\n" );
2988     return SUCCESS;
2989 }
```

4.1.4.31 static int IDI_CMD__SPI_Config_Chip_Select_Behavior (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2717 of file idi.c.

References spi_cfg::chip_select_behavior, IDI_CSB_BUFFER, IDI_CSB_SOFTWARE, IDI_CSB_UINT16, IDI_CSB_↩UINT8, SPI_Configuration_Get(), SPI_Configuration_Set(), and strcmpi.

```
2718 { /*  idi spi mode [0/1/2/3/software/buffer/uint8/uint16] */
2719     int      error_code;
2720 //  int      csb;
2721     struct   spi_cfg cfg;
2722
2723     /* pull current configuration from the hardware -- allows for warm restore so to speak */
2724     error_code = SPI_Configuration_Get( &cfg );
2725     if ( error_code ) return error_code;
2726
```

```
2727     if ( argc < 1 )
2728     { /* read */
2729         printf( "SPI CSB: ");
2730         switch ( cfg.chip_select_behavior )
2731         {
2732             case IDI_CSB_SOFTWARE:  printf( "software" );   break;
2733             case IDI_CSB_BUFFER:  printf( "buffer"   );   break;
2734             case IDI_CSB_UINT8:        printf( "uint8"    );   break;
2735             case IDI_CSB_UINT16:  printf( "uint16"   );   break;
2736             default:                printf( "undefined");   break;
2737         }
2738         printf( "\n" );
2739     }
2740     else
2741     { /* write */
2742         if      ( 0 == strcmpi( "software", argv[0] ) ) cfg.chip_select_behavior = 0;
2743         else if ( 0 == strcmpi( "buffer",   argv[0] ) ) cfg.chip_select_behavior = 1;
2744         else if ( 0 == strcmpi( "uint8",    argv[0] ) ) cfg.chip_select_behavior = 2;
2745         else if ( 0 == strcmpi( "uint16",   argv[0] ) ) cfg.chip_select_behavior = 3;
2746         else
2747         {
2748             cfg.chip_select_behavior = (SPI_CSB_ENUM) strtol( argv[0], NULL, 0 );
2749             switch ( cfg.chip_select_behavior )
2750             {
2751                 case IDI_CSB_SOFTWARE:
2752                 case IDI_CSB_BUFFER:
2753                 case IDI_CSB_UINT8:
2754                 case IDI_CSB_UINT16:
2755                     break;
2756                 default:
2757                     error_code = -EC_SPI_CSB_OUT_OF_RANGE;
2758                     break;
2759             }
2760         }
2761          if ( error_code ) return error_code;
2762         /* commit configuration to hardware */
2763         error_code = SPI_Configuration_Set( &cfg );
2764         if ( error_code ) return error_code;
2765         printf( "OK\n" );
2766     }
2767     return SUCCESS;
2768 }
```

4.1.4.32 static int IDI_CMD__SPI_Config_Clock_Hz (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2499 of file idi.c.

References spi_cfg::clock_hz, SPI_Calculate_Clock(), SPI_Configuration_Get(), and SPI_Configuration_Set().

```
2500 { /*  idi spi clk [<freq_hz>] */
2501     int      error_code;
2502     double   clock_request_hz;
2503     double   clock_actual_hz;
2504     double   error;
2505     uint16_t hci;
2506     struct   spi_cfg cfg;
2507
2508     /* pull current configuration from the hardware -- allows for warm restore so to speak */
2509     error_code = SPI_Configuration_Get( &cfg );
2510     if ( error_code ) return error_code;
2511
2512     if ( argc < 1 )
2513     { /* read */
2514         printf( "SPI CLK: %f hz\n", cfg.clock_hz );
2515     }
2516     else
2517     { /* write */
2518         clock_request_hz = atof( argv[0] );
2519         error_code = SPI_Calculate_Clock( clock_request_hz, &clock_actual_hz, &error, &
      hci );
```

```
2520         if ( error_code ) return error_code;
2521         //cfg.half_clock_interval = hci;
2522         /* commit configuration to hardware */
2523         cfg.clock_hz = clock_actual_hz;
2524         error_code   = SPI_Configuration_Set( &cfg );
2525         if ( error_code ) return error_code;
2526         printf( "OK\n" );
2527     }
2528     return SUCCESS;
2529 }
```

4.1.4.33 static int IDI_CMD__SPI_Config_End_Cycle_Delay_Sec (int *argc,* char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2536 of file idi.c.

References spi_cfg::end_delay_ns, spi_cfg::half_clock_interval, SPI_Calculate_End_Cycle_Delay(), SPI_Calculate_↩ Half_Clock_Interval_Sec(), SPI_Configuration_Get(), and SPI_Configuration_Set().

```
2537 { /*  idi spi ecd [<time sec>] */
2538     int      error_code;
2539     double   request_sec;
2540     double   actual_sec;
2541     double   error;
2542     uint8_t  ecd;
2543     struct   spi_cfg cfg;
2544 
2545     /* pull current configuration from the hardware -- allows for warm restore so to speak */
2546     error_code = SPI_Configuration_Get( &cfg );
2547     if ( error_code ) return error_code;
2548 
2549     if ( argc < 1 )
2550     { /* read */
2551         printf( "SPI ECD: %g sec\n", ( cfg.end_delay_ns * 1.0e-9 ) );
2552     }
2553     else
2554     { /* write */
2555         request_sec = atof( argv[0] );
2556         error_code = SPI_Calculate_End_Cycle_Delay(
      SPI_Calculate_Half_Clock_Interval_Sec( cfg.half_clock_interval ),
2557                                                  request_sec,
2558                                                  &actual_sec,
2559                                                  &error,
2560                                                  &ecd
2561                                                 );
2562         if ( error_code ) return error_code;
2563         cfg.end_delay_ns = actual_sec * 1.0e9;
2564         /* commit configuration to hardware */
2565         error_code = SPI_Configuration_Set( &cfg );
2566         if ( error_code ) return error_code;
2567         printf( "OK\n" );
2568     }
2569     return SUCCESS;
2570 }
```

4.1.4.34 static int IDI_CMD__SPI_Config_Get (int *argc,* char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2477 of file idi.c.

References SPI_Configuration_Get(), and SPI_Report_Configuration_Text().

```
2478 { /*  idi spi cfg */
2479     int     error_code;
2480     struct  spi_cfg cfg;
2481     (void)  argc;
2482     (void)  argv;
2483 
2484     error_code = SPI_Configuration_Get( &cfg );
2485     if ( error_code ) return error_code;
2486 
2487     error_code = SPI_Report_Configuration_Text( &cfg, stdout );
2488     if ( error_code ) return error_code;
2489 
2490     printf( "\n" );
2491     return SUCCESS;
2492 }
```

4.1.4.35 static int IDI_CMD__SPI_Config_Mode (int *argc,* char * *argv[]*) [static]

CPOL CPHA MODE 0 0 0 1 0 1 2 1 0 3 1 1.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2586 of file idi.c.

References spi_cfg::sclk_phase, spi_cfg::sclk_polarity, SPI_Configuration_Get(), and SPI_Configuration_Set().

```
2587 { /*  idi spi mode [0/1/2/3] */
2588     int      error_code;
2589     int      mode;
2590     struct   spi_cfg cfg;
2591 
2592     /* pull current configuration from the hardware -- allows for warm restore so to speak */
2593     error_code = SPI_Configuration_Get( &cfg );
2594     if ( error_code ) return error_code;
2595 
2596     if ( argc < 1 )
2597     { /* read */
2598         if      ( (false == cfg.sclk_polarity ) && (false == cfg.sclk_phase) ) mode = 0;
2599         else if ( (false == cfg.sclk_polarity ) && (true  == cfg.sclk_phase) ) mode = 1;
2600         else if ( (true  == cfg.sclk_polarity ) && (false == cfg.sclk_phase) ) mode = 2;
2601         else if ( (true  == cfg.sclk_polarity ) && (true  == cfg.sclk_phase) ) mode = 3;
2602         printf( "SPI MODE: %d\n", mode );
2603     }
2604     else
2605     { /* write */
2606         mode = (int) strtol( argv[0], NULL, 0 );
2607         switch ( mode )
2608         {
2609             case 0: cfg.sclk_polarity = false;  cfg.sclk_phase = false;     break;
2610             case 1: cfg.sclk_polarity = false;  cfg.sclk_phase = true;      break;
2611             case 2: cfg.sclk_polarity = true;   cfg.sclk_phase = false;     break;
2612             case 3: cfg.sclk_polarity = true;   cfg.sclk_phase = true;      break;
2613         }
2614         /* commit configuration to hardware */
2615         error_code = SPI_Configuration_Set( &cfg );
2616         if ( error_code ) return error_code;
2617         printf( "OK\n" );
2618     }
2619     return SUCCESS;
2620 }
```

4.1.4.36 static int IDI_CMD__SPI_Config_SDI_Polarity (int *argc,* char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2627 of file idi.c.

References spi_cfg::sdi_polarity, SPI_Configuration_Get(), SPI_Configuration_Set(), and String_To_Bool().

```
2628 { /*  idi spi sdi [<true/1/false/0>] */
2629     int      error_code;
2630 //  BOOL     value;
2631     struct   spi_cfg cfg;
2632
2633     /* pull current configuration from the hardware -- allows for warm restore so to speak */
2634     error_code = SPI_Configuration_Get( &cfg );
2635     if ( error_code ) return error_code;
2636
2637     if ( argc < 1 )
2638     { /* read */
2639         printf( "SPI SDI POLARITY: %s\n", cfg.sdi_polarity ? "true" : "false" );
2640     }
2641     else
2642     { /* write */
2643         cfg.sdi_polarity = String_To_Bool( argv[0] );
2644         /* commit configuration to hardware */
2645         error_code = SPI_Configuration_Set( &cfg );
2646         if ( error_code ) return error_code;
2647         printf( "OK\n" );
2648     }
2649     return SUCCESS;
2650 }
```

4.1.4.37 static int IDI_CMD__SPI_Config_SDIO_Wrap (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2687 of file idi.c.

References spi_cfg::sdio_wrap, SPI_Configuration_Get(), SPI_Configuration_Set(), and String_To_Bool().

```
2688 { /*  idi spi wrap [<true/1/false/0>] */
2689     int      error_code;
2690 //  BOOL     value;
2691     struct   spi_cfg cfg;
2692
2693     /* pull current configuration from the hardware -- allows for warm restore so to speak */
2694     error_code = SPI_Configuration_Get( &cfg );
2695     if ( error_code ) return error_code;
2696
2697     if ( argc < 1 )
2698     { /* read */
2699         printf( "SPI wrap: %s\n", cfg.sdio_wrap ? "true" : "false" );
2700     }
2701     else
2702     { /* write */
2703         cfg.sdio_wrap = String_To_Bool( argv[0] );
2704         /* commit configuration to hardware */
2705         error_code = SPI_Configuration_Set( &cfg );
2706         if ( error_code ) return error_code;
2707         printf( "OK\n" );
2708     }
2709     return SUCCESS;
2710 }
```

4.1.4.38 static int IDI_CMD__SPI_Config_SDO_Polarity (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2657 of file idi.c.

References spi_cfg::sdo_polarity, SPI_Configuration_Get(), SPI_Configuration_Set(), and String_To_Bool().

```
2658 { /*  idi spi sdo [<true/1/false/0>] */
2659     int      error_code;
2660 //  BOOL     value;
2661     struct   spi_cfg cfg;
2662
2663     /* pull current configuration from the hardware -- allows for warm restore so to speak */
2664     error_code = SPI_Configuration_Get( &cfg );
2665     if ( error_code ) return error_code;
2666
2667     if ( argc < 1 )
2668     { /* read */
2669         printf( "SPI SDO POLARITY: %s\n", cfg.sdo_polarity ? "true" : "false" );
2670     }
2671     else
2672     { /* write */
2673         cfg.sdo_polarity = String_To_Bool( argv[0] );
2674         /* commit configuration to hardware */
2675         error_code = SPI_Configuration_Set( &cfg );
2676         if ( error_code ) return error_code;
2677         printf( "OK\n" );
2678     }
2679     return SUCCESS;
2680 }
```

4.1.4.39 static int IDI_CMD__SPI_Data (int *argc,* char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2813 of file idi.c.

References HEX_DUMP_BYTES_PER_LINE, Hex_Dump_Line(), SPI_Data_Write_Read(), and SPI_FIFO_SIZE.

```
2814 { /*  idi spi data [byte] [character] ... */
2815     int                 error_code;
2816     size_t              index;
2817     size_t              count;
2818     size_t              transfer_count; /* */
2819     size_t              lines;
2820     uint8_t *           bp;             /* buffer pointer */
2821     uint8_t             tx_buffer[SPI_FIFO_SIZE];
2822     uint8_t             rx_buffer[SPI_FIFO_SIZE];
2823
2824
2825     if ( argc < 1 ) return -EC_PARAMETER;
2826
2827     index = 1;
2828     transfer_count = argc - 1;
2829     if ( transfer_count > SPI_FIFO_SIZE )
2830     {
2831         transfer_count = SPI_FIFO_SIZE;
2832         printf( "Warning: ignored %d values\n", argc - 1 - SPI_FIFO_SIZE );
2833     }
2834     count = transfer_count;
2835     while ( count != 0 )
2836     {
2837         tx_buffer[index-1] = (uint8_t) strtol( argv[index], NULL, 0 );
2838         count--; index++;
2839     }
2840     error_code = SPI_Data_Write_Read( sizeof( uint8_t ), transfer_count, tx_buffer,
     transfer_count, rx_buffer );
2841     if ( error_code ) return error_code;
2842
```

```
2843     lines = transfer_count / HEX_DUMP_BYTES_PER_LINE;
2844     if ( 0 == ( transfer_count - lines * HEX_DUMP_BYTES_PER_LINE ) ) lines = lines -
     1;
2845     for ( index = 0; index <= lines; index++ )
2846     {
2847         bp = &(rx_buffer[index * HEX_DUMP_BYTES_PER_LINE]);
2848         if ( transfer_count < HEX_DUMP_BYTES_PER_LINE )
2849         {
2850             Hex_Dump_Line( 0, transfer_count, bp, stdout );
2851         }
2852         else
2853         {
2854             Hex_Dump_Line( 0, HEX_DUMP_BYTES_PER_LINE, bp, stdout );
2855         }
2856     }
2857     return SUCCESS;
2858 }
```

4.1.4.40 static int IDI_CMD__SPI_FIFO (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2865 of file idi.c.

References HEX_DUMP_BYTES_PER_LINE, Hex_Dump_Line(), SPI_Commit(), SPI_FIFO_Read(), SPI_FIFO_SIZE, SPI_FIFO_Write(), SPI_Status_Read_FIFO_Is_Not_Empty(), SPI_Status_Write_FIFO_Is_Not_Empty(), strcmpi, and String_To_Bool().

```
2866 { /*  idi spi data [byte] [character] ... */
2867     int                 error_code;
2868     int                 index;
2869     int                 count;
2870     int                 read_count; /* */
2871     uint8_t             data_temp;
2872     uint8_t             tx_buffer[SPI_FIFO_SIZE];
2873     uint8_t             rx_buffer[SPI_FIFO_SIZE];
2874
2875
2876     read_count = 0;
2877     if ( argc < 1 )
2878     {
2879         read_count = SPI_FIFO_SIZE;
2880     }
2881     else
2882     {
2883         if      ( 0 == strcmpi( "rx", argv[0] ) )
2884         {
2885             if ( argc > 1)
2886             {
2887                 if ( 0 == strcmpi( "all", argv[1] ) ) read_count =
     SPI_FIFO_SIZE;
2888                 else read_count = (int) strtol( argv[1], NULL, 0 );
2889             }
2890         }
2891         else if ( 0 == strcmpi( "tx", argv[0] ) )
2892         {
2893             char * endptr;
2894
2895             index = 1;
2896             count = argc - 1;
2897             if ( count > SPI_FIFO_SIZE )
2898             {
2899                 count = SPI_FIFO_SIZE;
2900                 printf( "Warning: ignored %d values\n", argc - 1 - SPI_FIFO_SIZE );
2901             }
2902             while ( count > 0 )
2903             {
2904                 data_temp = (uint8_t) strtol( argv[index], &endptr, 0 );
2905                 if ( endptr == argv[index] ) data_temp = (uint8_t) argv[index][0];
2906                 tx_buffer[index-1] = data_temp;
2907                 count--; index++;
```

```
2908             }
2909             error_code = SPI_FIFO_Write( (void *) tx_buffer, sizeof( uint8_t ), count, NULL )
    ;
2910             if ( error_code ) return error_code;
2911 
2912             printf( "OK\n" );
2913             return error_code;
2914         }
2915         else if ( 0 == strcmpi( "commit", argv[0] ) )
2916         {
2917             if ( argc > 2 )
2918             {
2919                 if ( String_To_Bool( argv[1] ) ) SPI_Commit( 0xFF );
2920                 else                             SPI_Commit( 0x00 );
2921 
2922                 printf( "OK\n" );
2923                 error_code = SUCCESS;
2924             }
2925             else
2926             {
2927 //TODO: add ability to read-back the chip select under certain conditions.
2928                 error_code = -EC_PARAMETER_MISSING;
2929             }
2930             return error_code;
2931         }
2932     }
2933 
2934     /* wait for transmit data to empty out */
2935     while ( SPI_Status_Write_FIFO_Is_Not_Empty() ) { /* do nothing */ }
2936 
2937     if ( read_count > 0 )
2938     {
2939         int       lines;
2940         uint8_t * bp; /* buffer pointer */
2941 
2942         if ( SPI_Status_Read_FIFO_Is_Not_Empty() )
2943         {
2944             error_code = SPI_FIFO_Read( (void *) rx_buffer, sizeof( uint8_t ), read_count,
    NULL );
2945             if ( error_code ) return error_code;
2946 
2947             lines = read_count / HEX_DUMP_BYTES_PER_LINE;
2948             if ( 0 == ( read_count - lines * HEX_DUMP_BYTES_PER_LINE ) ) lines =
    lines - 1;
2949             for ( index = 0; index <= lines; index++ )
2950             {
2951                 bp = &(rx_buffer[index * HEX_DUMP_BYTES_PER_LINE]);
2952                 if ( read_count < HEX_DUMP_BYTES_PER_LINE )
2953                 {
2954                     Hex_Dump_Line( 0, read_count, bp, stdout );
2955                 }
2956                 else
2957                 {
2958                     Hex_Dump_Line( 0, HEX_DUMP_BYTES_PER_LINE, bp, stdout );
2959                 }
2960             }
2961         }
2962         else
2963         {
2964             printf( "FIFO Empty\n" );
2965         }
2966     }
2967     return SUCCESS;
2968 }
```

4.1.4.41 static int IDI_CMD__SPI_ID (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2461 of file idi.c.

References SPI_ID_Get().

```
2462 { /*  idi spi id */
2463     uint16_t    id;
2464     (void)      argc;
2465     (void)      argv;
2466
2467     SPI_ID_Get( &id );
2468     printf( "SPI ID: 0x%04X\n", id );
2469     return SUCCESS;
2470 }
```

4.1.4.42 static int IDI_CMD__SPI_Status (int *argc*, char * *argv[]*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 2775 of file idi.c.

References SPI_Report_Status_Text(), SPI_Status_Read(), SPI_Status_Write(), and strcmpi.

```
2776 { /*  idi spi status [rx] [tx] ...  */
2777     int                 error_code;
2778     int                 index;
2779     struct spi_status     status;
2780
2781     if ( argc < 1 )
2782     {
2783         error_code = SPI_Status_Read( &status );
2784         if ( error_code ) return error_code;
2785         SPI_Report_Status_Text( &status, stdout );
2786     }
2787     else
2788     {
2789         for ( index = 0; index < argc; index++ )
2790         {
2791             if ( 0 == strcmpi( "rx", argv[index] ) )
2792             {
2793                 error_code = SPI_Status_Read( &status );
2794                 if ( error_code ) return error_code;
2795                 SPI_Report_Status_Text( &status, stdout );
2796             }
2797             else if ( 0 == strcmpi( "tx", argv[index] ) )
2798             {
2799                 error_code = SPI_Status_Write( &status );
2800                 if ( error_code ) return error_code;
2801                 SPI_Report_Status_Text( &status, stdout );
2802             }
2803         }
2804     }
2805     return SUCCESS;
2806 }
```

4.1.4.43 int IDI_Command_Line_Digital_Input (int *argc*, char * *argv[]*)

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3425 of file idi.c.

References command_line::cmd_fnc, command_line::name, and strcmpi.

Referenced by IDI_Command_Line_Main().

```
3426 { /*  idi din <channel/all>  */
3427     int         error_code;
3428     int         index;
```

```
3429     int         argc_new;
3430     char **     argv_new;
3431     char *      endptr;
3432     //int           channel;
3433
3434     if ( argc < 1 ) return -EC_NOT_FOUND;
3435
3436     error_code = -EC_SYNTAX;
3437
3438     //channel = (int) strtol( argv[0], &endptr, 0 );
3439     strtol( argv[0], &endptr, 0 ); /* just want to know where it fails */
3440     if ( argv[0] != endptr )
3441     {  /* assume channel number */
3442         error_code = (* idi_cmd_din[0].cmd_fnc )( argc, argv );
3443     }
3444     else
3445     { /* otherwise a normal command */
3446         index = 0;
3447         while ( NULL != idi_cmd_din[index].cmd_fnc )
3448         {
3449             if ( 0 == strcmpi( idi_cmd_din[index].name, argv[0] )  )
3450             {
3451                 argv_new = &(argv[1]);
3452                 argc_new = argc - 1;
3453                 error_code = (* idi_cmd_din[index].cmd_fnc )( argc_new, argv_new );
3454                 break;
3455             }
3456             index++;
3457         }
3458     }
3459     return error_code;
3460 }
```

4.1.4.44 int IDI_Command_Line_FRAM (int *argc*, char * *argv[]*)

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3232 of file idi.c.

References command_line::cmd_fnc, command_line::name, and strcmpi.

Referenced by IDI_Command_Line_Main().

```
3233 {
3234     int         error_code;
3235     int         index;
3236     int         argc_new;
3237     char **     argv_new;
3238
3239     if ( argc < 1 ) return -EC_NOT_FOUND;
3240
3241     index = 0;
3242     while ( NULL != idi_cmd_fram[index].cmd_fnc )
3243     {
3244         if ( 0 == strcmpi( idi_cmd_fram[index].name, argv[0] )  )
3245         {
3246             argv_new = &(argv[1]);
3247             argc_new = argc - 1;
3248             error_code = (* idi_cmd_fram[index].cmd_fnc )( argc_new, argv_new );
3249             break;
3250         }
3251         index++;
3252     }
3253     return error_code;
3254 }
```

4.1.4.45 int IDI_Command_Line_Main (int *argc,* char * *argv[]*)

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.

Parameters

in	*argc*	number of arguments including the executable file name
in	*argv*	list of string arguments lex'd from the command line

Returns

SUCCESS (0) if no errors encountered, otherwise errors are reported as a negative value.

Definition at line 3651 of file idi.c.

References IDI_Command_Line_Digital_Input(), IDI_Command_Line_FRAM(), IDI_Command_Line_Register_↩ Transaction(), IDI_Command_Line_Set(), IDI_Command_Line_SPI(), IDI_Register_Report_CSV(), and strcmpi.

Referenced by main().

```
3652 {
3653     int     error_code;
3654     int     index;
3655     int     argc_new;
3656     char ** argv_new;
3657 
3658     index = 0; /* offset into the string list */
3659 
3660     if ( argc > 0 /* at least one */ )
3661     {
3662         if      ( 0 == strcmpi( "set", argv[index] ) )
3663         { /* idi set .... */
3664             index++;
3665             argv_new = &(argv[index]);
3666             argc_new = argc - index;
3667             error_code = IDI_Command_Line_Set( argc_new, argv_new );
3668             if ( error_code ) goto IDI_COMMAND_LINE_MAIN_TERMINATE;
3669         }
3670         else if ( 0 == strcmpi( "dump", argv[index] ) )
3671         { /* idi dump .... */
3672             FILE * fd_out;
3673             index++;
3674             if ( argc > 2 )
3675             {
3676                 fd_out = fopen( argv[index], "w" );
3677                 if ( NULL == fd_out ) fd_out = stdout;
3678             }
3679             else
3680             {
3681                 fd_out = stdout;
3682             }
3683             error_code = IDI_Register_Report_CSV(
    definitions, fd_out );
3684 
3685             if ( (argc > 2) && (NULL != fd_out) && (stdout != fd_out) )
3686             {
3687                 fclose( fd_out );
3688             }
3689             if ( error_code ) goto IDI_COMMAND_LINE_MAIN_TERMINATE;
3690         }
3691         else if ( 0 == strcmpi( "spi", argv[index] ) )
3692         { /* idi spi .... */
3693             // idi spi id
3694             // idi spi
3695             index++;
3696             argv_new = &(argv[index]);
3697             argc_new = argc - index;
3698             error_code = IDI_Command_Line_SPI( argc_new, argv_new );
3699             if ( error_code ) goto IDI_COMMAND_LINE_MAIN_TERMINATE;
3700         }
3701         else if ( 0 == strcmpi( "fram", argv[index] ) )
3702         { /* idi fram .... */
3703             index++;
3704             argv_new = &(argv[index]);
3705             argc_new = argc - index;
3706             error_code = IDI_Command_Line_FRAM( argc_new, argv_new );
3707             if ( error_code ) goto IDI_COMMAND_LINE_MAIN_TERMINATE;
3708         }
3709         else if ( 0 == strcmpi( "din", argv[index] ) )
```

```
3710         { /* idi din .... */
3711             index++;
3712             argv_new = &(argv[index]);
3713             argc_new = argc - index;
3714             error_code = IDI_Command_Line_Digital_Input( argc_new, argv_new )
     ;
3715             if ( error_code ) goto IDI_COMMAND_LINE_MAIN_TERMINATE;
3716         }
3717         else
3718         {
3719             argv_new = &(argv[index]);
3720             argc_new = argc - index;
3721             error_code = IDI_Command_Line_Register_Transaction(
     argc_new, argv_new );
3722             if ( error_code ) goto IDI_COMMAND_LINE_MAIN_TERMINATE;
3723         }
3724     }
3725 IDI_COMMAND_LINE_MAIN_TERMINATE:
3726     return error_code;
3727 }
```

4.1.4.46 int IDI_Command_Line_Register_Transaction (int *argc*, char * *argv[]*)

Either reads or writes a register using the form: idi <register acronym>=""> [

].

If

is not include, then it is assumed to be a read. If value is included, then a write to the specified register is made.

This function uses the definitions[] array which is global and built from IDI_REGISTER_SET_DEFINITION macro which is a nicely organized register list.

Parameters

in	*argc*	number of arguments including the executable file name
in	*argv*	list of string arguments lex'd from the command line

Returns

SUCCESS (0) if no errors encountered, otherwise errors are reported as a negative value.

Definition at line 3488 of file idi.c.

References IO_Read_U8(), IO_Write_U8(), REG_DIR_NONE, and strcmpi.

Referenced by IDI_Command_Line_Main().

```
3489 {
3490     int error_code;
3491     int index;
3492     int found;
3493
3494     if ( argc < 1 ) return -EC_NOT_FOUND;
3495
3496     found = -1;
3497     index =  0;
3498     while ( definitions[index].direction != REG_DIR_NONE )
3499     {
3500         if ( 0 == strcmpi( definitions[index].acronym, argv[0] )  )
3501         {
3502             found = index;
3503             break;
3504         }
3505         index++;
3506     }
3507
3508     if ( found < 0 )
3509     {
```

```
3510         //printf( "ER: \n" );
3511         return  -EC_NOT_FOUND;
3512     }
3513
3514     if ( argc < 2 )
3515     { /* read operation */
3516         uint8_t value;
3517         error_code = IO_Read_U8( definitions[index].symbol, &value );
3518         if ( SUCCESS == error_code )
3519         {
3520             printf( "RD: %s=0x%02X\n",  definitions[index].acronym, value );
3521         }
3522     }
3523     else
3524     {
3525         uint8_t value;
3526         value = (uint8_t) strtol( argv[1], NULL, 0 );
3527         error_code = IO_Write_U8( definitions[index].symbol, value );
3528         if ( SUCCESS == error_code )
3529         {
3530             printf( "WR: %s=0x%02X\n", definitions[index].acronym, value );
3531         }
3532     }
3533
3534     return SUCCESS;
3535 }
```

4.1.4.47 int IDI_Command_Line_Set (int *argc*, char * *argv[]*)

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3616 of file idi.c.

References command_line::cmd_fnc, command_line::name, and strcmpi.

Referenced by IDI_Command_Line_Main().

```
3617 {
3618     int         error_code;
3619     int         index;
3620     int         argc_new;
3621     char **     argv_new;
3622
3623     if ( argc < 1 ) return -EC_NOT_FOUND;
3624
3625     index = 0;
3626     while ( NULL != idi_cmd_fram[index].cmd_fnc )
3627     {
3628         if ( 0 == strcmpi( idi_cmd_main[index].name, argv[0] )  )
3629         {
3630             argv_new = &(argv[1]);
3631             argc_new = argc - 1;
3632             error_code = (* idi_cmd_main[index].cmd_fnc )( argc_new, argv_new );
3633             break;
3634         }
3635         index++;
3636     }
3637     return error_code;
3638 }
```

4.1.4.48 int IDI_Command_Line_SPI (int *argc*, char * *argv[]*)

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3017 of file idi.c.

References command_line::cmd_fnc, command_line::name, and strcmpi.

Referenced by IDI_Command_Line_Main().

```
3018 {
3019     int         error_code;
3020     int         index;
3021     int         argc_new;
3022     char **     argv_new;
3023
3024     if ( argc < 1 ) return -EC_NOT_FOUND;
3025
3026     index = 0;
3027     while ( NULL != idi_cmd_spi[index].cmd_fnc )
3028     {
3029         if ( 0 == strcmpi( idi_cmd_spi[index].name, argv[0] )  )
3030         {
3031             argv_new = &(argv[1]);
3032             argc_new = argc - 1;
3033             error_code = (* idi_cmd_spi[index].cmd_fnc )( argc_new, argv_new );
3034             break;
3035         }
3036         index++;
3037     }
3038     return error_code;
3039 }
```

4.1.4.49 static int IDI_DIN_Channel_Get (size_t *channel*, BOOL * *value*) [static]

Obtains and reports a single digital input channel.

Parameters

in	*channel*	channel to be read out.
out	*value*	Pointer to the boolean value to be set based on the digital input value

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 992 of file idi.c.

References IDI_DIN_GROUP_SIZE, IDI_DIN_SHIFT_RIGHT, and IO_Read_U8().

Referenced by IDI_CMD__DIN_Channel().

```
993 {
994     size_t  group;
995     size_t  bit;
996     uint8_t reg_value;
997
998     group = channel >> IDI_DIN_SHIFT_RIGHT;
999     bit   = channel - group * IDI_DIN_GROUP_SIZE;
1000
1001     IO_Read_U8( IDI_DI_GROUP0 + group, &reg_value );
1002
1003     if ( 0 != ( reg_value & ( 0x01 << bit ) ) ) *value = true;
1004     else                                        *value = false;
1005
1006     return SUCCESS;
1007 }
```

4.1.4.50 static int IDI_DIN_Group_Get (size_t *group*, uint8_t * *value*) [static]

Reads the selected digital input port (8-bits).

Parameters

in	*group*	the group, range is 0 to 5.
out	*value*	pointer to the destination for the data read out

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1018 of file idi.c.

References IO_Read_U8().

Referenced by IDI_CMD__DIN_All(), and IDI_CMD__DIN_Group().

```
1019 {
1020     IO_Read_U8( IDI_DI_GROUP0 + group, value );
1021     return SUCCESS;
1022 }
```

4.1.4.51 int IDI_DIN_ID_Get (uint16_t * *id*)

Obtains the DIN component (or board ID in this case) ID number.

Parameters

out	*id*	The 16-bit ID number

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 955 of file idi.c.

References ID_DIN, and IO_Read_U8().

Referenced by IDI_CMD__DIN_ID(), and IDI_DIN_IsNotPresent().

```
956 {
957     uint8_t     lsb, msb;
958
959     IO_Read_U8( IDI_ID_LSB, &lsb );
960     IO_Read_U8( IDI_ID_MSB, &msb );
961     *id = ( ((uint16_t) msb) << 8 ) | ((uint16_t) lsb);
962 #if defined( ID_ALWAYS_REPORT_AS_GOOD )

963     *id = ID_DIN;
964 #endif

965     return SUCCESS;
966 }
```

4.1.4.52 BOOL IDI_DIN_IsNotPresent (void)

Determines if the DIN component and/or board is present. Returns true if not present (i.e. error).

Returns

A zero is returned if present, otherwise a 1 is returned.

Definition at line 975 of file idi.c.

References ID_DIN, and IDI_DIN_ID_Get().

```
976 {
977     uint16_t id;
978     IDI_DIN_ID_Get( &id );
979     if (ID_DIN == id ) return 0;
980     return 1;
981 }
```

4.1.4.53 void IDI_Help (FILE * *out*)

Outputs a help listing to the user.

Definition at line 3746 of file idi.c.

References command_line::help, command_line::name, and idi_dataset::svn_revision_string.

Referenced by main().

```
3747 {
3748     int index;
3749
3750     fprintf( out, "\n" );
3751     fprintf( out, "Isolated Digital Input Test Code\n" );
3752     fprintf( out, "Apex Embedded Systems\n" );
3753     fprintf( out, "Revision: %s\n", idi_dataset.svn_revision_string );
3754
3755     fprintf( out, "\n" );
3756     fprintf( out, "help - outputs help information\n" );
3757
3758     fprintf( out, "\n" );
3759     fprintf( out, "loop - any command below can run in a loop until key pressed\n" );
3760
3761     fprintf( out, "\n" );
3762     fprintf( out, "dump - outputs register information in a comma delimited format\n" );
3763
3764     fprintf( out, "\n" );
3765     fprintf( out, "set  - Set/Get main parameters.\n" );
3766     index = 0;
3767     while ( NULL != idi_cmd_main[index].help )
3768     {
3769     fprintf( out, "       %8s - %s\n", idi_cmd_main[index].name,
      idi_cmd_main[index].help );
3770     index++;
3771     }
3772
3773     fprintf( out, "\n" );
3774     fprintf( out, "spi  - SPI related functions\n" );
3775     index = 0;
3776     while ( NULL != idi_cmd_spi[index].help )
3777     {
3778     fprintf( out, "       %8s - %s\n", idi_cmd_spi[index].name,
      idi_cmd_spi[index].help );
3779     index++;
3780     }
3781
3782     fprintf( out, "\n" );
3783     fprintf( out, "fram - FRAM related functions\n" );
3784     index = 0;
3785     while ( NULL != idi_cmd_fram[index].help )
3786     {
3787     fprintf( out, "       %8s - %s\n", idi_cmd_fram[index].name,
      idi_cmd_fram[index].help );
3788     index++;
3789     }
3790
3791     fprintf( out, "\n" );
```

```
3792     fprintf( out, "din  - Digital input related functions\n" );
3793     index = 0;
3794     while ( NULL != idi_cmd_din[index].help )
3795     {
3796     fprintf( out, "       %8s - %s\n", idi_cmd_din[index].name,
      idi_cmd_din[index].help );
3797     index++;
3798     }
3799     fprintf( out, "\n" );
3800 }
```

4.1.4.54 int IDI_Initialization (void)

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.

Returns

SUCCESS (0) if no errors encountered, otherwise errors are reported as a negative value.

Definition at line 3836 of file idi.c.

References idi_dataset::bank_previous, idi_dataset::base_address, IDI_BANK_0, idi_svn_revision_string, and idi_↩ dataset::svn_revision_string.

Referenced by main().

```
3837 {
3838     FILE * fd;
3839
3840     /* restore the data set, if we can otherwise initialize with defaults */
3841     fd = fopen( "idi_init.bin", "r" );
3842     if ( NULL == fd )
3843     { /* defaults */
3844         memset( &idi_dataset, 0, sizeof(struct idi_dataset) );
3845         idi_dataset.base_address        = 0xff00;
3846         idi_dataset.bank_previous       = IDI_BANK_0;
3847     }
3848     else
3849     { /* read in dataset */
3850         fread( &idi_dataset, 1, sizeof( struct idi_dataset ), fd );
3851         fclose( fd );
3852     }
3853     idi_dataset.svn_revision_string =
      idi_svn_revision_string;
3854
3855     return SUCCESS;
3856 }
```

4.1.4.55 int IDI_Register_Report_CSV (const struct reg_definition * *table*, FILE * *out*)

Outputs a human readable CSV to the desired output file or stdout.

Parameters

in	*table*	register definition table or data structure array
in	*out*	destination of the human readable text to specified file or terminal.

Returns

SUCCESS (0) if no errors encountered, otherwise errors are reported as a negative value.

Definition at line 485 of file idi.c.

References 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, REG_DIR_NONE, REG_DIR_READ, REG_DIR_READ_WRITE, and REG_DIR_WRITE.

Referenced by IDI_Command_Line_Main().

```
486 {
487     int index = 0;
488
489     fprintf( out, "\"acronym\",\"symbol\",\"bank\",\"direction\",\"physical_offset\"\n" );
490     do
491     {
492         fprintf( out, "\"%s\",", table[index].acronym     );
493         fprintf( out, "\"%s\",", table[index].symbol_name );
494
495         switch( table[index].bank )
496         {
497         case IDI_BANK_0:          fprintf( out, "IDI_BANK_0" );           break;
498         case IDI_BANK_1:          fprintf( out, "IDI_BANK_1" );           break;
499         case IDI_BANK_2:          fprintf( out, "IDI_BANK_2" );           break;
500         case IDI_BANK_3:          fprintf( out, "IDI_BANK_3" );           break;
501         case IDI_BANK_4:          fprintf( out, "IDI_BANK_4" );           break;
502         case IDI_BANK_5:          fprintf( out, "IDI_BANK_5" );           break;
503         case IDI_BANK_6:          fprintf( out, "IDI_BANK_6" );           break;
504         case IDI_BANK_7:          fprintf( out, "IDI_BANK_7" );           break;
505         case IDI_BANK_NONE:            fprintf( out, "IDI_BANK_NONE" );       break;
506         case IDI_BANK_UNDEFINED:  fprintf( out, "IDI_BANK_UNDEFINED" );   break;
507         }
508         fprintf( out, "," );
509
510         switch( table[index].direction )
511         {
512             case REG_DIR_NONE:          fprintf( out, "REG_DIR_NONE" );         break;
513             case REG_DIR_READ:          fprintf( out, "REG_DIR_READ" );         break;
514             case REG_DIR_WRITE:            fprintf( out, "REG_DIR_WRITE" );        break;
515             case REG_DIR_READ_WRITE:  fprintf( out, "REG_DIR_READ_WRITE" );   break;
516         }
517         fprintf( out, "," );
518
519         fprintf( out, "\"%d\"", table[index].physical_offset );
520         fprintf( out, "\n\r" ); /* separate so we have flexibility to re-organize columns */
521         index++;
522     } while ( definitions[index].direction != REG_DIR_NONE );
523
524     return SUCCESS;
525 }
```

4.1.4.56 static const char* IDI_Symbol_Name_Bank (IDI_BANK_ENUM *bank*) [static]

Outputs a human readable CSV to the desired output file or stdout.

Parameters

in	*bank*	the bank enumerated value written to the bank register.

Returns

a string that describes the selected bank.

Definition at line 456 of file idi.c.

References 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_symbol_names, and IDI_BANK_UNDEFINED.

Referenced by IO_Read_U8(), and IO_Write_U8().

```
457 {
458     int index;
459
460     switch( bank )
461     {
462         case IDI_BANK_0:          index = 0;  break;
463         case IDI_BANK_1:          index = 1;  break;
464         case IDI_BANK_2:          index = 2;  break;
465         case IDI_BANK_3:          index = 3;  break;
466         case IDI_BANK_4:          index = 4;  break;
467         case IDI_BANK_5:          index = 5;  break;
468         case IDI_BANK_6:          index = 6;  break;
469         case IDI_BANK_7:          index = 7;  break;
470         case IDI_BANK_NONE:            index = 8;  break;
471         case IDI_BANK_UNDEFINED:  index = 9;  break;
472     }
473     return idi_bank_symbol_names[index];
474 }
```

4.1.4.57 int IDI_Termination (void)

Runs upon application exit. It saves the idi_dataset data structure.

Returns

SUCCESS (0) if no errors encountered, otherwise errors are reported as a negative value.

Definition at line 3809 of file idi.c.

Referenced by main().

```
3810 {
3811     FILE * fd;
3812
3813     /* save the data set */
3814     fd = fopen( "idi_init.bin", "w" );
3815     if ( NULL == fd )
3816     { /* defaults */
3817         return -EC_INIT_FILE;
3818     }
3819     else
3820     { /* read in dataset */
3821         fwrite( &idi_dataset, 1, sizeof( struct idi_dataset ), fd );
3822         fclose( fd );
3823     }
3824     return SUCCESS;
3825 }
```

4.1.4.58 static BOOL IO_Direction_IsNotValid (IDI_REG_ENUM *location*, REG_DIR_ENUM *direction*) [static]

Looks up in the register definitions list for the ports possible read/write directions.

Parameters

in	*location*	the enumerated register symbol. The enumerated symbol is composed of offset and bank information used to determine the final address information.

in	*direction*	The desired direction that is to run

Returns

A false is returned if the direction is valid, otherwise a true is returned

Definition at line 718 of file idi.c.

References REG_DIR_NONE.

Referenced by IO_Read_U8(), and IO_Write_U8().

```
719 {
720     int index;
721     index =  0;
722     while ( definitions[index].direction != REG_DIR_NONE )
723     {
724         if ( location == definitions[index].symbol  )
725         {
726             if ( definitions[index].direction & direction ) return false;
727             else                                            return true;
728         }
729         index++;
730     }
731     return true;
732 }
```

4.1.4.59 static char* IO_Get_Symbol_Name (IDI_REG_ENUM *location*) [static]

Translates a register enumerated symbol into a string that is the same as the enumerated symbol used throughout this code base.

Parameters

in	*location*	The enumerated symbol representing the register.

Returns

a human readable string of the symbol.

Definition at line 692 of file idi.c.

References REG_DIR_NONE, and reg_definition::symbol_name.

Referenced by IO_Read_U8(), and IO_Write_U8().

```
693 {
694     int index;
695     index =  0;
696     while ( definitions[index].direction != REG_DIR_NONE )
697     {
698         if ( location == definitions[index].symbol  )
699         {
700             return definitions[index].symbol_name;
701         }
702         index++;
703     }
704     return NULL;
705 }
```

4.1.4.60 void IO_Read_U16_Address_Fixed (IDI_REG_ENUM *location*, uint16_t * *value*)

Reads uint16_t from I/O ports in a uint8_t succession to the same address location. Macros are used to guide the target implementation. In this case, bus width (which we typcially refer to the port width, which is different than register width) is assumed to be byte (uint8_t) wide.

Parameters

in	*location*	the enumerated register symbol. The enumerated symbol is composed of offset and bank information used to determine the final address information.
in	*value*	The pointer to the destination of the read uint16_t value.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 931 of file idi.c.

References IO_Read_U8().

```
932 {
933 //TODO: assumes little endian.
934     uint8_t lsb, msb;
935     IO_Read_U8( (IDI_REG_ENUM)(((int) location) + 0), &lsb );
936     IO_Read_U8( (IDI_REG_ENUM)(((int) location) + 0), &msb );
937     *value = ( ((uint16_t) msb) << 8 ) | ( ((uint16_t) lsb) & 0xFF );
938 }
```

4.1.4.61 void IO_Read_U16_Address_Increment (IDI_REG_ENUM *location*, uint16_t * *value*)

Reads uint16_t from I/O ports in a uint8_t succession incrementing the offset address. Macros are used to guide the target implementation. In this case, bus width (which we typcially refer to the port width, which is different than register width) is assumed to be byte (uint8_t) wide.

Parameters

in	*location*	the enumerated register symbol. The enumerated symbol is composed of offset and bank information used to determine the final address information.
in	*value*	The pointer to the destination of the read uint16_t value.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 887 of file idi.c.

References IO_Read_U8().

```
888 {
889 //TODO: assumes little endian.
890     uint8_t lsb, msb;
891     IO_Read_U8( (IDI_REG_ENUM)(((int) location) + 0), &lsb );
892     IO_Read_U8( (IDI_REG_ENUM)(((int) location) + 1), &msb );
893     *value = ( ((uint16_t) msb) << 8 ) | ( ((uint16_t) lsb) & 0xFF );
894 }
```

4.1.4.62 int IO_Read_U8 (IDI_REG_ENUM *location*, uint8_t * *value*)

Reads uint8_t from I/O port. Macros are used to guide the target implementation.

Parameters

in	*location*	the enumerated register symbol. The enumerated symbol is composed of offset and bank information used to determine the final address information.
in	*value*	The pointer to the destination of the read uint8_t value.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 802 of file idi.c.

References idi_dataset::bank_previous, idi_dataset::base_address, IDI_BANK_NONE, IDI_Symbol_Name_Bank(), IO_Direction_IsNotValid(), IO_Get_Symbol_Name(), idi_dataset::io_report, idi_dataset::io_simulate, REG_DIR_READ, REG_LOCATION_BANK_GET, and REG_LOCATION_OFFSET_GET.

Referenced by IDI_Command_Line_Register_Transaction(), IDI_DIN_Channel_Get(), IDI_DIN_Group_Get(), IDI_DIN_ID_Get(), IO_Read_U16_Address_Fixed(), IO_Read_U16_Address_Increment(), SPI_Configuration_Chip_Select_Behavior_Get(), SPI_Configuration_Chip_Select_Behavior_Set(), SPI_Configuration_Get(), SPI_Data_Write_Read_Helper(), SPI_FIFO_Read(), SPI_ID_Get(), SPI_Status_Read(), SPI_Status_Read_FIFO_Is_Not_Empty(), SPI_Status_Read_FIFO_Status(), SPI_Status_Write(), SPI_Status_Write_FIFO_Is_Full(), SPI_Status_Write_FIFO_Is_Not_Empty(), and SPI_Status_Write_FIFO_Status().

```
803 {
804     uint8_t     bank;
805     int         offset;
806     int         address;
807 
808 #if defined( IDI_IO_DIRECTION_TEST )

809     if ( IO_Direction_IsNotValid( location, REG_DIR_READ ) )
810     {
811         printf( "IO_Read_U8: %s, error in direction\n", IO_Get_Symbol_Name( location ) );
812         return -EC_DIRECTION;
813     }
814 #endif

815     bank   = (uint8_t) REG_LOCATION_BANK_GET( location );
816     if ( ( IDI_BANK_NONE != bank ) && ( bank != idi_dataset.
      bank_previous ) )
817     {   /* write to bank register only if different -- don't bother even checking it, will take too much
       time. */
818         offset = (int) REG_LOCATION_OFFSET_GET( IDI_BANK );
819         address = ((int) idi_dataset.base_address) + offset;
820         idi_dataset.bank_previous = bank;
821         if ( !idi_dataset.io_simulate )
822         {
823 #if defined( __MSDOS__ )

824         outportb( address, bank );
825 #endif

826         }
827         if ( ( idi_dataset.io_report ) || ( idi_dataset.
      io_simulate ) )
828         {
829         printf( "IO_Read_U8: %s, address = 0x%04X, bank = %s\n",
      IO_Get_Symbol_Name( IDI_BANK ), address, IDI_Symbol_Name_Bank( bank )
       );
830         }
831     }
832     offset = (int) REG_LOCATION_OFFSET_GET(location);
833     address = ((int) idi_dataset.base_address) + offset;
834     if ( !idi_dataset.io_simulate )
835     {
836 #if defined( __MSDOS__ )

837     *value = inportb( address );
838 #endif

839     }
840     if ( ( idi_dataset.io_report ) || ( idi_dataset.
```

```
      io_simulate ) )
841     {
842     printf( "IO_Read_U8: %s, address = 0x%04X, ", IO_Get_Symbol_Name( location ), address
       );
843 #if defined( __MSDOS__ )

844     printf( "value = 0x%02X\n", *value );
845 #else

846     printf( "value = unknown\n" );
847 #endif

848     }
849     return SUCCESS;
850 }
```

4.1.4.63 void IO_Write_U16_Address_Fixed (IDI_REG_ENUM *location,* uint16_t *value*)

Writes uint16_t to I/O ports in a uint8_t succession to the same address location. Macros are used to guide the target implementation. In this case, bus width (which we typcially refer to the port width, which is different than register width) is assumed to be byte (uint8_t) wide.

Parameters

in	*location*	the enumerated register symbol. The enumerated symbol is composed of offset and bank information used to determine the final address information.
in	*value*	The uint16_t value to be written to the I/O register

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 910 of file idi.c.

References IO_Write_U8().

```
911 {
912 //TODO: assumes little endian.
913     IO_Write_U8( (IDI_REG_ENUM)(((int) location) + 0), (uint8_t)(     value      & 0
      xFF ) );
914     IO_Write_U8( (IDI_REG_ENUM)(((int) location) + 0), (uint8_t)( ( value >> 8 ) & 0
      xFF ) );
915 }
```

4.1.4.64 void IO_Write_U16_Address_Increment (IDI_REG_ENUM *location,* uint16_t *value*)

Writes uint16_t to I/O ports in a uint8_t succession incrementing the offset address. Macros are used to guide the target implementation. In this case, bus width (which we typcially refer to the port width, which is different than register width) is assumed to be byte (uint8_t) wide.

Parameters

in	*location*	the enumerated register symbol. The enumerated symbol is composed of offset and bank information used to determine the final address information.

in	*value*	The uint16_t value to be written to the I/O register

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 866 of file idi.c.

References IO_Write_U8().

```
867 {
868 //TODO: assumes little endian.
869     IO_Write_U8( (IDI_REG_ENUM)(((int) location) + 0), (uint8_t)(   value        & 0
      xFF ) );
870     IO_Write_U8( (IDI_REG_ENUM)(((int) location) + 1), (uint8_t)( ( value >> 8 ) & 0
      xFF ) );
871 }
```

4.1.4.65 int IO_Write_U8 (IDI_REG_ENUM *location*, uint8_t *value*)

Writes uint8_t to I/O port. Macros are used to guide the target implementation.

Parameters

in	*location*	the enumerated register symbol. The enumerated symbol is composed of offset and bank information used to determine the final address information.
in	*value*	The data to be written out.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 746 of file idi.c.

References idi_dataset::bank_previous, idi_dataset::base_address, IDI_BANK_NONE, IDI_Symbol_Name_Bank(), IO_Direction_IsNotValid(), IO_Get_Symbol_Name(), idi_dataset::io_report, idi_dataset::io_simulate, REG_DIR_WRITE, REG_LOCATION_BANK_GET, and REG_LOCATION_OFFSET_GET.

Referenced by IDI_Command_Line_Register_Transaction(), IO_Write_U16_Address_Fixed(), IO_Write_U16_Address_Increment(), SPI_Commit(), SPI_Configuration_Chip_Select_Behavior_Set(), SPI_Configuration_Set(), SPI_Data_Write_Read_Helper(), and SPI_FIFO_Write().

```
747 {
748     uint8_t     bank;
749     uint8_t     offset;
750     uint16_t    address;
751
752 #if defined( IDI_IO_DIRECTION_TEST )

753     if ( IO_Direction_IsNotValid( location, REG_DIR_WRITE ) )
754     {
755         printf( "IO_Write_U8: %s, error in direction\n", IO_Get_Symbol_Name( location ) )
      ;
756         return -EC_DIRECTION;
757     }
758 #endif

759     bank   = (uint8_t) REG_LOCATION_BANK_GET( location );
760     if ( ( IDI_BANK_NONE != bank ) && ( bank != idi_dataset.
      bank_previous ) )
761     {   /* write to bank register only if different -- don't bother even checking it, will take too much
       time. */
762         offset = (uint8_t) REG_LOCATION_OFFSET_GET( IDI_BANK );
```

```
763         address = idi_dataset.base_address + offset;
764         idi_dataset.bank_previous = bank;
765         if ( !idi_dataset.io_simulate )
766         {
767 #if defined( __MSDOS__ )

768         outportb( address, bank );
769 #endif

770         }
771         if ( ( idi_dataset.io_report ) || ( idi_dataset.
      io_simulate ) )
772         {
773         printf( "IO_Write_U8: %s, address = 0x%04X, bank = %s\n",
      IO_Get_Symbol_Name( IDI_BANK ), address, IDI_Symbol_Name_Bank( bank )
       );
774         }
775     }
776     offset = (uint8_t) REG_LOCATION_OFFSET_GET(location);
777     address = idi_dataset.base_address + offset;
778     if ( !idi_dataset.io_simulate )
779     {
780 #if defined( __MSDOS__ )

781     outportb( address, value );
782 #endif

783     }
784     if ( ( idi_dataset.io_report ) || ( idi_dataset.
      io_simulate ) )
785     {
786     printf( "IO_Write_U8: %s, address = 0x%04X, value = 0x%02X\n",
      IO_Get_Symbol_Name( location ), address, value );
787     }
788     return SUCCESS;
789 }
```

4.1.4.66 int main (int *argc*, char * *argv[]*)

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.

Parameters

in	*argc*	number of arguments including the executable file name
in	*argv*	list of string arguments lex'd from the command line

Returns

SUCCESS (0) if no errors encountered, otherwise errors are reported as a negative value.

Definition at line 3871 of file idi.c.

References Character_Get(), EC_Code_To_Human_Readable(), IDI_Command_Line_Main(), IDI_Help(), IDI_↩ Initialization(), IDI_Termination(), and strcmpi.

```
3872 {
3873     //int index;
3874     //int count;
3875     int     error_code;
3876     int     index;
3877     int     argc_new;
3878     char ** argv_new;
3879
3880 /* used only for Win7 debugging sessions with cygwin */
3881 #if(1)

3882     setvbuf(stdout, NULL, _IONBF, 0);
3883     setvbuf(stderr, NULL, _IONBF, 0);
3884 #endif
```

```
3885
3886     error_code = IDI_Initialization();
3887     if ( error_code ) goto Main_Termination;
3888 //      printf( "Hello\n");
3889 //      return SUCCESS;
3890
3891 #if(0)

3892     count = argc;
3893     index = 0;      /* OK, zero value is name of executable file  */
3894     while ( count > 0 )
3895     {
3896         printf( "index = %d, str = <%s>\n", index, argv[index] );
3897         index++;
3898         count--;
3899     }
3900 #endif

3901
3902     index = 1;
3903     if ( argc > 1 )
3904     {
3905         if      ( 0 == strcmpi( "help", argv[index]) )
3906         {
3907             IDI_Help( stdout );
3908         }
3909         else if ( 0 == strcmpi( "loop", argv[index]) )
3910         { /* loop until key is pressed */
3911             if ( argc > 2 )
3912             {
3913                 index++;
3914                 while ( !Character_Get(NULL) )
3915                 { /* assumes that all functions utilize arguments in read only fashion */
3916                     argv_new = &(argv[index]);
3917                     argc_new = argc - index;
3918                     error_code = IDI_Command_Line_Main( argc_new, argv_new );
3919                     if ( error_code ) goto Main_Termination_Error_Codes;
3920                 }
3921             }
3922         }
3923         else
3924         {
3925             argv_new = &(argv[index]);
3926             argc_new = argc - index;
3927             error_code = IDI_Command_Line_Main( argc_new, argv_new );
3928             if ( error_code ) goto Main_Termination_Error_Codes;
3929         }
3930     }
3931     else
3932     { /* produce help */
3933         IDI_Help( stdout );
3934     }
3935
3936
3937 Main_Termination:
3938     IDI_Termination();
3939     return error_code;
3940
3941 Main_Termination_Error_Codes:
3942     IDI_Termination();
3943     printf( "ERROR: %d, %s\n",  error_code, EC_Code_To_Human_Readable( error_code
      ) );
3944     return error_code;
3945 }
```

4.1.4.67 int SPI_Calculate_Clock (double *clock_request_hz*, double * *clock_actual_hz*, double * *error*, uint16_t * *hci*)

Computes the SPI clock half clock register value given a requested SPI clock frequency. It will also produce a 'report' indicating the actual value (due to integer resolution) as well as a computed error between requested and actual. The eror can be used to determine whether timing constraints are met.

Parameters

in	*clock_request_hz*	Request clock frequency in Hertz. Example: 1.0e6 is 1MHz.
in	*clock_actual_hz*	Actual computed frequency. If this pointer is NULL, then it is not output.
out	*error*	Error between requested and actual. If this pointer is NULL, then it is not output.
out	*hci*	Half clock register value computed. If this pointer is NULL, then it is not output.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1121 of file idi.c.

References SPI_Calculate_Half_Clock().

Referenced by IDI_CMD__SPI_Config_Clock_Hz(), and SPI_Configuration_Set().

```
1126 {
1127     int      error_code;
1128     double   half_clock_request_sec;
1129     double   half_clock_actual_sec;
1130     double   error_internal;
1131     double   scratch;
1132     uint16_t hci_internal;
1133
1134     half_clock_request_sec = 1.0 / ( 2.0 * clock_request_hz );
1135
1136     error_code = SPI_Calculate_Half_Clock( half_clock_request_sec,
1137                                            &half_clock_actual_sec,
1138                                            &error_internal,
1139                                            &hci_internal
1140                                          );
1141     if ( error_code ) return error_code;
1142
1143     /* compute actual frequency */
1144     scratch = 1.0 / ( 2.0 * half_clock_actual_sec );
1145     if ( NULL != error           ) *error           = ( scratch - clock_request_hz ) / clock_request_hz;
1146     if ( NULL != clock_actual_hz ) *clock_actual_hz = scratch;
1147     if ( NULL != hci             ) *hci             = (uint16_t) hci_internal;
1148     return SUCCESS;
1149 }
```

4.1.4.68 int SPI_Calculate_End_Cycle_Delay (double *spi_half_clock_interval_sec*, double *delay_request_sec*, double * *delay_actual_sec*, double * *error*, uint8_t * *ecd*)

Computes the time delay at the end of each byte transmitted. It will only output the parameters whose pointers are not NULL.

Parameters

in	*spi_half_clock_interval_sec*	Computed half clock interval in seconds
in	*delay_request_sec*	Requested time delay in seconds
out	*delay_actual_sec*	Pointer to actual time delay computed, if not NULL.
out	*error*	Pointer to error value computed, if not NULL.
out	*ecd*	Pointer to the computed end-cycle-delay, if not NULL.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1178 of file idi.c.

References CLOCK_PERIOD_SEC.

Referenced by IDI_CMD__SPI_Config_End_Cycle_Delay_Sec(), and SPI_Configuration_Set().

```
1184 {
1185     //double        delay_between_words_sec;
1186     double      scratch;
1187     int         ecd_temp;
1188
1189     /* delay_sec = CLOCK_PERIOD_SEC * 4 + ECD * spi_half_clock_interval_sec  */
1190     scratch = ( delay_request_sec - 4.0 * CLOCK_PERIOD_SEC ) / spi_half_clock_interval_sec;
1191     ecd_temp = (int) scratch;
1192
1193     if ( ( ecd_temp > 255 ) || ( ecd_temp < 0 ) ) return -EC_SPI_ECD_OUT_OF_RANGE;
1194
1195     /* compute actual */
1196     scratch = CLOCK_PERIOD_SEC * 4.0 + ((double) ecd_temp) * spi_half_clock_interval_sec;
1197     if ( NULL != error            ) *error            = ( scratch - delay_request_sec ) / delay_request_sec
     ;
1198     if ( NULL != delay_actual_sec ) *delay_actual_sec = scratch;
1199     if ( NULL != ecd              ) *ecd              = (uint8_t) ecd_temp;
1200     return SUCCESS;
1201 }
```

4.1.4.69 int SPI_Calculate_Half_Clock (double *half_clock_request_sec*, double * *half_clock_actual_sec*, double * *error*, uint16_t * *hci*)

Computes the half clock register value given a requested time interval. It will also produce a 'report' indicating the actual value (due to integer resolution) as well as a computed error between requested and actual. The error can be used to determine whether timing constraints are met.

Parameters

in	*half_clock_request_sec*	Request time interval in seconds. Example: 20.0e-6 is 20uS.
in	*half_clock_actual_sec*	Actual computed time. If this pointer is NULL, then it is not output.
out	*error*	Error between requested and actual. If this pointer is NULL, then it is not output.
out	*hci*	Half clock register value computed. If this pointer is NULL, then it is not output.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1084 of file idi.c.

References CLOCK_PERIOD_SEC.

Referenced by SPI_Calculate_Clock().

```
1089 {
1090     double      scratch;
1091     int         hci_temp;
1092
1093     /* spi_half_clock_interval_sec = CLOCK_PERIOD_SEC * ( 4.0 + ( (double) hci ) ) */
```

```
1094     scratch = ( half_clock_request_sec / CLOCK_PERIOD_SEC ) - 4.0;
1095     hci_temp = (int) scratch;
1096
1097     if ( ( hci_temp > 4095 ) || ( hci_temp < 0 ) ) return -EC_SPI_HALF_CLOCK_OUT_OF_RANGE;
1098
1099     /* compute actual */
1100     scratch = CLOCK_PERIOD_SEC * ( 4.0 + ((double) hci_temp) );
1101     if ( NULL != error                ) *error                = ( scratch - half_clock_request_sec ) /
     half_clock_request_sec;
1102     if ( NULL != half_clock_actual_sec ) *half_clock_actual_sec = scratch;
1103     if ( NULL != hci                  ) *hci                  = (uint16_t) hci_temp;
1104     return SUCCESS;
1105 }
```

4.1.4.70 double SPI_Calculate_Half_Clock_Interval_Sec (uint16_t *half_clock_interval*)

Computes the half clock interval in seconds given the value from the half clock interval register.

Parameters

in	*half_clock_interval*	Half clock interval register value

Returns

The time value as a double in units of seconds.

Definition at line 1158 of file idi.c.

References CLOCK_PERIOD_SEC.

Referenced by IDI_CMD__SPI_Config_End_Cycle_Delay_Sec(), and SPI_Configuration_Set().

```
1159 {
1160     double half_clock_interval_sec;
1161     half_clock_interval_sec = CLOCK_PERIOD_SEC * ( 4.0 + ((double) half_clock_interval ) );
1162     return half_clock_interval_sec;
1163 }
```

4.1.4.71 int SPI_Commit (uint8_t *chip_select*)

Sets/Clears the chip select or used to commit the transmit/write FIFO to the spi interface. The mode of operation is dependent on the chip_select_behavior.

Parameters

in	*chip_select*	Used to write to the SCS_COMMIT bit. Its behavior is dependent on the chip_select_behavior.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1606 of file idi.c.

References IO_Write_U8(), and SPI_IsNotPresent().

Referenced by FRAM__Memory_Read(), FRAM__Memory_Write(), IDI_CMD__SPI_Commit(), IDI_CMD__SPI_FIFO(), and SPI_Data_Write_Read_Helper().

```
1607 {
1608     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
1609     IO_Write_U8( SPI_COMMIT, chip_select );
1610     return SUCCESS;
1611 }
```

4.1.4.72 int SPI_Configuration_Chip_Select_Behavior_Get (SPI_CSB_ENUM * *chip_select_behavior*)

Extracts the chip select behavior from the SPI configuration register.

Parameters

out	*chip_select_behavior*	pointer to the destination of the value obtained.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1211 of file idi.c.

References IO_Read_U8(), and SPI_IsNotPresent().

Referenced by SPI_Data_Write_Read().

```
1212 {
1213     uint8_t     scratch;
1214
1215     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
1216
1217     IO_Read_U8( SPI_CONFIG, &scratch );
1218
1219     *chip_select_behavior = (SPI_CSB_ENUM) ( scratch >> 4 ) & 0x07;
1220     return SUCCESS;
1221 }
```

4.1.4.73 int SPI_Configuration_Chip_Select_Behavior_Set (SPI_CSB_ENUM *chip_select_behavior*)

Sets the chip select behavior to the SPI configuration register.

Parameters

in	*chip_select_behavior*	enumerated value to be written to the register.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1231 of file idi.c.

References IO_Read_U8(), IO_Write_U8(), and SPI_IsNotPresent().

Referenced by FRAM__Memory_Read(), FRAM__Memory_Write(), FRAM__Read_ID(), FRAM__Read_Status_Register(), FRAM__Write_Disable(), FRAM__Write_Enable_Latch_Set(), FRAM__Write_Status_Register(), and SPI_Data_Write_Read().

```
1232 {
1233     uint8_t     scratch;
1234
```

```
1235     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
1236
1237     IO_Read_U8( SPI_CONFIG, &scratch );
1238
1239     scratch &= 0x70;
1240     scratch |= (uint8_t) ( ( chip_select_behavior & 0x07 ) << 4 );
1241
1242     IO_Write_U8( SPI_CONFIG, scratch );
1243     return SUCCESS;
1244 }
```

4.1.4.74 int SPI_Configuration_Get (struct spi_cfg * *cfg*)

Obtains the SPI configuration from the hardware.

Parameters

out	*cfg*	SPI configuration data structure or data set

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1309 of file idi.c.

References spi_cfg::chip_select_behavior, spi_cfg::clock_hz, CLOCK_PERIOD_SEC, spi_cfg::end_cycle_delay, spi_cfg::end_delay_ns, spi_cfg::half_clock_interval, IO_Read_U8(), spi_cfg::sclk_phase, spi_cfg::sclk_polarity, spi_cfg::sdi_polarity, spi_cfg::sdio_wrap, spi_cfg::sdo_polarity, idi_dataset::spi_cfg, and SPI_IsNotPresent().

Referenced by IDI_CMD__SPI_Config_Chip_Select_Behavior(), IDI_CMD__SPI_Config_Clock_Hz(), IDI_CMD__SPI_Config_End_Cycle_Delay_Sec(), IDI_CMD__SPI_Config_Get(), IDI_CMD__SPI_Config_Mode(), IDI_CMD__SPI_Config_SDI_Polarity(), IDI_CMD__SPI_Config_SDIO_Wrap(), and IDI_CMD__SPI_Config_SDO_Polarity().

```
1310 {
1311     uint8_t     scratch;
1312
1313     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
1314
1315     IO_Read_U8( SPI_CONFIG, &scratch );
1316     cfg->chip_select_behavior   = (SPI_CSB_ENUM) ( scratch >> 4 ) & 0x07;
1317     cfg->sclk_polarity         = (BOOL) ( scratch & 0x01 );
1318     cfg->sclk_phase               = (BOOL) ( scratch & 0x02 );
1319     cfg->sdi_polarity           = (BOOL) ( scratch & 0x04 );
1320     cfg->sdo_polarity           = (BOOL) ( scratch & 0x08 );
1321     cfg->sdio_wrap             = (BOOL) ( scratch & 0x80 );
1322
1323     IO_Read_U8( SPI_HCI_LSB, &scratch );
1324     cfg->half_clock_interval = (uint16_t) scratch;
1325     IO_Read_U8( SPI_HCI_MSB, &scratch );
1326     cfg->half_clock_interval |= ( (uint16_t) scratch) << 8;
1327
1328     IO_Read_U8( SPI_ECD, &(cfg->end_cycle_delay) );
1329
1330     cfg->clock_hz = 1.0 / ( 2.0 * CLOCK_PERIOD_SEC * ( 4.0 + ((double) cfg->
      half_clock_interval) ) );
1331     cfg->end_delay_ns = 1.0e9 * CLOCK_PERIOD_SEC * 4.0 + 0.5 * ((double) cfg->
      end_cycle_delay) / cfg->clock_hz;
1332
1333     memcpy( &(idi_dataset.spi_cfg), &cfg, sizeof( struct
      spi_cfg ) );
1334     return SUCCESS;
1335 }
```

4.1.4.75 int SPI_Configuration_Initialize (struct spi_cfg * *cfg*)

Initializes the SPI configuration data structure.

[in] cfg Pointer to the SPI configuration data structure to be initialized

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1408 of file idi.c.

References spi_cfg::chip_select_behavior, spi_cfg::clock_hz, spi_cfg::end_cycle_delay, spi_cfg::end_delay_ns, spi_cfg::half_clock_interval, spi_cfg::sclk_phase, spi_cfg::sclk_polarity, spi_cfg::sdi_polarity, spi_cfg::sdio_wrap, and spi_cfg::sdo_polarity.

```
1409 {
1410     cfg->sdio_wrap                = false;
1411     cfg->sdo_polarity           = false;
1412     cfg->sdi_polarity           = false;
1413     /*  Mode    CPOL    CPHA

1414      *   0       0       0

1415      *   1       0       1

1416      *   2       1       0

1417      *   3       1       1

1418      */
1419     cfg->sclk_phase           = false;   /* the FRAM uses SPI Mode 0 or 3 */
1420     cfg->sclk_polarity            = false;
1421
1422     cfg->chip_select_behavior   = false;
1423     cfg->end_cycle_delay         = 0;        /* shortest delay possible      */
1424     cfg->half_clock_interval     = 0;        /* shortest interval possible   */
1425
1426     cfg->clock_hz               = 0.0;
1427     cfg->end_delay_ns           = 0.0;
1428
1429     return SUCCESS;
1430 }
```

4.1.4.76 int SPI_Configuration_Set (struct spi_cfg * *cfg*)

Commits the configuration data structure to the hardware.

Parameters

in	*cfg*	The software configuration data structure to be committed to hardware

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1254 of file idi.c.

References spi_cfg::chip_select_behavior, spi_cfg::clock_hz, spi_cfg::end_cycle_delay, spi_cfg::end_delay_ns, spi_cfg::half_clock_interval, IO_Write_U8(), spi_cfg::sclk_phase, spi_cfg::sclk_polarity, spi_cfg::sdi_polarity, spi_cfg::sdio_wrap, spi_cfg::sdo_polarity, SPI_Calculate_Clock(), SPI_Calculate_End_Cycle_Delay(), SPI_Calculate_Half_Clock_Interval_Sec(), idi_dataset::spi_cfg, and SPI_IsNotPresent().

Referenced by IDI_CMD__SPI_Config_Chip_Select_Behavior(), IDI_CMD__SPI_Config_Clock_Hz(), IDI_CMD__SPI_Config_End_Cycle_Delay_Sec(), IDI_CMD__SPI_Config_Mode(), IDI_CMD__SPI_Config_SDI_Polarity(), IDI_CMD__SPI_Config_SDIO_Wrap(), and IDI_CMD__SPI_Config_SDO_Polarity().

```
1255 {
```

```
1256     int         error_code;
1257     double      scratch;
1258     double      hci_sec;     /* half clock interval in seconds */
1259     uint8_t     config;
1260
1261     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
1262
1263     config = (uint8_t) ( ( cfg->chip_select_behavior & 0x07 ) << 4 );
1264     if ( cfg->sclk_polarity ) config |= 0x01;
1265     if ( cfg->sclk_phase    ) config |= 0x02;
1266     if ( cfg->sdi_polarity  ) config |= 0x04;
1267     if ( cfg->sdo_polarity  ) config |= 0x08;
1268     if ( cfg->sdio_wrap     ) config |= 0x80;
1269
1270     IO_Write_U8( SPI_CONFIG, config );
1271
1272     if ( cfg->clock_hz > 0 )
1273     { /* compute half_clock_interval */
1274         //scratch = ( 1.0 - ( 8.0 * CLOCK_PERIOD_SEC * cfg->clock_hz ) ) / (  2.0 * CLOCK_PERIOD_SEC *
      cfg->clock_hz );
1275         error_code = SPI_Calculate_Clock( cfg->clock_hz, NULL, NULL, &(cfg->
     half_clock_interval) );
1276         if ( error_code ) return error_code;
1277     }
1278     hci_sec = SPI_Calculate_Half_Clock_Interval_Sec( cfg->
     half_clock_interval );
1279
1280     if ( cfg->end_delay_ns > 0 )
1281     {
1282         scratch    = cfg->end_delay_ns * 1.0e-9;
1283         error_code = SPI_Calculate_End_Cycle_Delay( hci_sec,               /*
      calculated half-clock interval     */
1284                                                     scratch,               /* requested end-delay interval
         */
1285                                                     NULL,                  /*  computed actual delay
         */
1286                                                     NULL,                  /* error between actual and
      desired   */
1287                                                     &(cfg->end_cycle_delay) /* computed
      count                      */
1288                                                    );
1289         if ( error_code ) return error_code;
1290     }
1291
1292     IO_Write_U8( SPI_HCI_LSB, (uint8_t)( cfg->half_clock_interval & 0xFF ) );
1293     IO_Write_U8( SPI_HCI_MSB, (uint8_t)( cfg->half_clock_interval >> 8   ) );
1294     IO_Write_U8( SPI_ECD, cfg->end_cycle_delay );
1295
1296     memcpy( &(idi_dataset.spi_cfg), &cfg, sizeof( struct
     spi_cfg ) );
1297
1298     return SUCCESS;
1299 }
```

4.1.4.77 int SPI_Data_Read (const void * *rx_buffer*, size_t *size*, size_t *rx_size*, FILE * *fd_log*)

Special case of Write/Read that has a function signature same as fread() or fwrite().

Parameters

in	*cfg*	pass in the configuration to be written to hardware.

Returns

a nonzero if successful, else return zero.
A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Parameters

size	object size
rx_size	object count
fd_log	set to NULL if no file logging

Definition at line 2023 of file idi.c.

References SPI_Data_Write_Read().

```
2028 {
2029     int error_code;
2030     error_code = SPI_Data_Write_Read(    size,
2031                                          0,      /* nothing to transmit */
2032                                          NULL,   /* nothing to transmit */
2033                                          rx_size,
2034                                          rx_buffer
2035                                     );
2036     if ( error_code ) return error_code;
2037
2038     if ( NULL != fd_log )
2039     {
2040         error_code = fwrite( rx_buffer, size, rx_size, fd_log );
2041     }
2042     return error_code;
2043 }
```

4.1.4.78 int SPI_Data_Write (const void * *tx_buffer*, size_t *size*, size_t *tx_count*, FILE * *fd_log*)

Special case of Write/Read that has a function signature same as fread() or fwrite().

Parameters

in	*cfg*	pass in the configuration to be written to hardware.

Returns

a nonzero if successful, else return zero.
A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Parameters

size	object size
tx_count	object count
fd_log	set to NULL if no file logging

Definition at line 1987 of file idi.c.

References SPI_Data_Write_Read().

```
1992 {
1993     int error_code;
1994     error_code = SPI_Data_Write_Read(    size,
1995                                          tx_count,
1996                                          tx_buffer,
1997                                          0,          /* nothing to receive */
1998                                          NULL        /* nothing to receive */
1999                                     );
2000     if ( error_code ) return error_code;
2001
2002     if ( NULL != fd_log )
2003     {
2004         error_code = fwrite( tx_buffer, size, tx_count, fd_log );
2005     }
2006     return error_code;
2007 }
```

4.1.4.79 int SPI_Data_Write_Read (size_t *size*, size_t *tx_count*, const void * *tx_buffer*, size_t *rx_size*, const void * *rx_buffer*)

This function will write/read virtually any kind of data with almost any kind of chips select wrapping surrounding the data.

Returns

a zero if successful, else return an error code.
A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Parameters

size	object size: 1=u8, 2=u16, 4=u16, 8=u64, 16=u128=SPI_FIFO_SIZE
tx_count	object count
rx_size	object count

Definition at line 1856 of file idi.c.

References IDI_CSB_BUFFER, IDI_CSB_SOFTWARE, IDI_CSB_UINT16, IDI_CSB_UINT8, SPI_Configuration_↩Chip_Select_Behavior_Get(), SPI_Configuration_Chip_Select_Behavior_Set(), SPI_Data_Write_Read_Helper(), SP↩I_FIFO_SIZE, and SPI_IsNotPresent().

Referenced by FRAM__Memory_Read(), FRAM__Memory_Write(), FRAM__Read_ID(), FRAM__Read_Status_↩Register(), FRAM__Write_Disable(), FRAM__Write_Enable_Latch_Set(), FRAM__Write_Status_Register(), IDI_CM↩D__SPI_Data(), SPI_Data_Read(), and SPI_Data_Write().

```
1862 {
1863     int                 error_code;
1864     int                 index;
1865     BOOL                active_tx;
1866     BOOL                active_rx;
1867     SPI_CSB_ENUM        csb_copy;
1868     SPI_CSB_ENUM        csb;
1869     BOOL                csb_buffer_mode_override;
1870
1871 /* TEST FOR VALIDITY OF PARAMETERS */
1872     /* see if there is anything to do */
1873     if ( ( NULL == tx_buffer ) && ( NULL == rx_buffer ) ) return SUCCESS;
1874
1875     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
1876
1877     /* initialize parameters */
1878     error_code = SPI_Configuration_Chip_Select_Behavior_Get( &csb
      );
1879     if ( error_code ) return error_code;
1880     //error_code = SPI_Status_Write( &status_tx );
1881     //if ( error_code ) return error_code;
1882     //error_code = SPI_Status_Read( &status_rx );
1883     //if ( error_code ) return error_code;
1884
1885     active_rx = false; /* assume that we toss any data to be read out */
1886     active_tx = false; /* assume that we have no valid data to write  */
1887     if ( NULL != tx_buffer ) active_tx = true;
1888     else                     tx_count  = rx_size;
1889     if ( NULL != rx_buffer ) active_rx = true;
1890     else                     rx_size   = tx_count;
1891
1892
1893     if ( IDI_CSB_UINT16 == csb )
1894     { /* test for even quantity of bytes to transceive */
1895         if ( ( tx_count & 0x01 ) || ( rx_size & 0x01 ) )
1896         { /* odd number of bytes detected for buffers */
1897             return -EC_SPI_BUFFER_SIZE_ODD;
1898         }
1899     }
1900
1901     csb_buffer_mode_override = false;
1902     if ( IDI_CSB_BUFFER == csb )
1903     { /* test for object size */
1904         if ( size > SPI_FIFO_SIZE ) return -EC_SPI_OBJECT_SIZE;
1905     }
1906     else
```

```
1907     {
1908         /* test object sizes */
1909         index = SPI_FIFO_SIZE;  /* assumed to be a 2^N number */
1910         while ( index > sizeof( uint16_t ) )
1911         {
1912             if ( size == ( size & index ) )
1913             {
1914                 csb_buffer_mode_override = true;
1915                 break;
1916             }
1917             index = index >> 1;
1918         }
1919         if ( ( size > 2 ) && ( false == csb_buffer_mode_override ) )
1920         {
1921             return -EC_SPI_OBJECT_SIZE; /* not a power of 2 */
1922         }
1923         else if ( true == csb_buffer_mode_override )
1924         { /* OK, go ahead and change to buffer mode */
1925             csb_copy   = csb;
1926             csb        = IDI_CSB_BUFFER;
1927             error_code = SPI_Configuration_Chip_Select_Behavior_Set
     ( csb );
1928         }
1929     }
1930
1931 /* PERFORM TRANSACTIONS */
1932     switch( csb )
1933     {
1934         case IDI_CSB_SOFTWARE:
1935         case IDI_CSB_UINT8:
1936         case IDI_CSB_UINT16:
1937             error_code = SPI_Data_Write_Read_Helper( size,
1938                                                     tx_count,
1939                                                     tx_buffer,
1940                                                     rx_size,
1941                                                     rx_buffer,
1942                                                     active_tx,
1943                                                     active_rx,
1944                                                     csb
1945                                                   );
1946             break;
1947         case IDI_CSB_BUFFER:
1948             for ( index = 0; index < tx_count; index++ )
1949             {
1950                 error_code = SPI_Data_Write_Read_Helper( size,
1951                                                          1,
1952                                                          active_tx ? &(((uint8_t *) tx_buffer)[index*size])
     : NULL,
1953                                                          1,
1954                                                          active_rx ? &(((uint8_t *) rx_buffer)[index*size])
     : NULL,
1955                                                          active_tx,
1956                                                          active_rx,
1957                                                          csb
1958                                                        );
1959             }
1960             if ( csb_buffer_mode_override )
1961             { /* restore to original csb */
1962                 error_code = SPI_Configuration_Chip_Select_Behavior_Set
     ( csb_copy );
1963             }
1964             break;
1965         default:
1966             return -EC_SPI_CSB_OUT_OF_RANGE;
1967             break;
1968     }
1969     return SUCCESS;
1970 }
```

4.1.4.80 static int SPI_Data_Write_Read_Helper (size_t *size*, size_t *tx_count*, const void * *tx_buffer*, size_t *rx_size*, const void * *rx_buffer*, BOOL *active_tx*, BOOL *active_rx*, SPI_CSB_ENUM *csb*) [static]

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Parameters

size	object size: u8 = 1, u16 = 2
tx_count	object count
rx_size	object count

Definition at line 1716 of file idi.c.

References IDI_CSB_SOFTWARE, IO_Read_U8(), IO_Write_U8(), SPI_Commit(), SPI_Status_Read_FIFO_Status(), and SPI_Status_Write_FIFO_Status().

Referenced by SPI_Data_Write_Read().

```
1725 {
1726     size_t  rx_bytes_available;
1727     BOOL    rx_empty;
1728     BOOL    tx_full;
1729     BOOL    tx_empty;
1730     size_t  tx_bytes_available;
1731     size_t  tx_index;
1732     size_t  rx_index;
1733     size_t  index;
1734     BOOL    commit_valid;
1735     uint8_t bit_bucket; /* tossed */
1736 
1737     /* verify size information */
1738     switch ( size )
1739     {
1740         case sizeof( uint8_t  ):
1741         case sizeof( uint16_t ):
1742             break; /* these sizes are OK */
1743         default:
1744             return -EC_SPI_OBJECT_SIZE;
1745             break;
1746     }
1747 
1748     /* initially need to make sure that both TX and RX are empty */
1749     do
1750     { //TODO: need a time out of some sort here and then return an error code.
1751         SPI_Status_Write_FIFO_Status( &tx_full, &tx_empty, &tx_bytes_available
     );
1752         if ( false == tx_empty ) SPI_Commit( 0xFF );
1753     } while ( false == tx_empty );
1754 
1755     do
1756     { //TODO: need a time out of some sort here and then return an error code.
1757         SPI_Status_Read_FIFO_Status( &rx_empty, &rx_bytes_available );
1758         if ( false == rx_empty ) IO_Read_U8( SPI_DATA, &bit_bucket  ); /* toss */
1759     } while ( false == rx_empty );
1760 
1761     tx_index     = 0;
1762     rx_index     = 0;
1763     commit_valid = false;
1764     while ( tx_index < tx_count )
1765     {
1766         /* get status simultaneously */
1767         SPI_Status_Write_FIFO_Status( &tx_full, &tx_empty, &tx_bytes_available
     );
1768         if ( (true == tx_full) && (false == commit_valid) )
1769         {
1770             if ( IDI_CSB_SOFTWARE != csb ) SPI_Commit( 0xFF ); /* does not matter
      what is written */
1771             commit_valid = true;
1772         }
1773         if ( (true == tx_empty) && (true == commit_valid) )
1774         {
1775             commit_valid = false; /* will need to restart the buffer transmission */
1776         }
1777         /* Write Data
1778          *
1779          */
1780         if ( tx_bytes_available > size )
1781         {   /* write data */
1782             if ( active_tx )
1783             {
1784             for ( index = 0; index < size; index++ ) IO_Write_U8( SPI_DATA, ((uint8_t *)
     tx_buffer)[tx_index] );
```

```
1785            }
1786            else
1787            {
1788            for ( index = 0; index < size; index++ ) IO_Write_U8( SPI_DATA, 0x00 ); /* send
     anything */
1789            }
1790            tx_index = tx_index + size;
1791        }
1792        /* Read Data

1793         *  This function will play catch up with respect to the transmit side.

1794         */
1795        /* get status simultaneously */
1796        SPI_Status_Read_FIFO_Status( &rx_empty, &rx_bytes_available );
1797        if ( rx_bytes_available >= size )
1798        { /* read data */
1799            if ( active_rx )
1800            {
1801            for ( index = 0; index < size; index++ ) IO_Read_U8( SPI_DATA, &(((uint8_t *)
      rx_buffer)[rx_index]) );
1802            }
1803            else
1804            {
1805            for ( index = 0; index < size; index++ ) IO_Read_U8( SPI_DATA, &bit_bucket ); /* toss
     */
1806            }
1807            rx_index = rx_index + size;
1808        }
1809    }
1810
1811    SPI_Status_Write_FIFO_Status( &tx_full, &tx_empty, &tx_bytes_available );
1812    if ( (false == commit_valid) && (false == tx_empty) )
1813    { /* data has not been transmitted yet */
1814        if ( IDI_CSB_SOFTWARE != csb ) SPI_Commit( 0xFF ); /* does not matter
     what is written */
1815        /* wait for the buffer to empty */
1816        do
1817        { //TODO: need a timeout and return error code if timeout exceeded.
1818            SPI_Status_Write_FIFO_Status( &tx_full, &tx_empty, &
      tx_bytes_available );
1819        } while ( false == tx_empty );
1820    }
1821
1822    /* retrieve the remaining read data and don't return until we are done */
1823    while ( rx_index != tx_index )
1824    { //TODO: timeout mechanism????
1825        SPI_Status_Read_FIFO_Status( &rx_empty, &rx_bytes_available );
1826        if ( rx_bytes_available >= size )
1827        { /* read data */
1828            if ( active_rx )
1829            {
1830            for ( index = 0; index < size; index++ ) IO_Read_U8( SPI_DATA, &(((uint8_t *)
      rx_buffer)[rx_index]) );
1831            }
1832            else
1833            {
1834            for ( index = 0; index < size; index++ ) IO_Read_U8( SPI_DATA, &bit_bucket ); /* toss
     */
1835            }
1836            rx_index = rx_index + size;
1837        }
1838    }
1839    return SUCCESS;
1840 }
```

4.1.4.81 int SPI_FIFO_Read (const void * *buffer*, size_t *size*, size_t *count*, FILE * *fd_log*)

Reads from the SPI receive/read data FIFO. It does not attempt to correlate the number of transmit bytes with receive bytes. Its purpose is more for low level hardware testing. Note that this function has a signature identical to the fread() function (i.e. make use of function pointers to guide sourcing of data).

Parameters

in	*buffer*	Buffer for the data destination.
in	*size*	Size of objects in bytes.
in	*count*	Number of objects to be read
out	*fd_log*	Optional log file to write the buffer too. If NULL, then no logging.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned indicating an error. In addition, a positive value is returned indicating the number of actual objects written.

Definition at line 1677 of file idi.c.

References IO_Read_U8(), SPI_FIFO_SIZE, SPI_IsNotPresent(), and SPI_Status_Read_FIFO_Status().

Referenced by IDI_CMD__SPI_FIFO().

```
1678 {
1679     int     error_code;       /* used primarily for debug purposes */
1680     size_t  bytes_available;
1681     BOOL    empty;
1682     BOOL    full;
1683     size_t  index;
1684     size_t  qty_objects;
1685     size_t  qty_bytes;
1686
1687     error_code = SUCCESS;
1688
1689     if ( (size * count) > SPI_FIFO_SIZE ) return -EC_PARAMETER;
1690
1691     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
1692
1693     SPI_Status_Read_FIFO_Status( &empty, &bytes_available );
1694
1695     qty_objects =  bytes_available / size; /* max number of objects that can be processed */
1696     if ( count > qty_objects ) count = qty_objects;
1697
1698     qty_bytes = count * size;
1699
1700     for ( index = 0; index < qty_bytes; index++ ) IO_Read_U8( SPI_DATA, &(((uint8_t *) buffer)[
      index]) );
1701
1702     if ( NULL != fd_log )
1703     {
1704         error_code = fwrite( buffer, size, qty_objects, fd_log );
1705     }
1706
1707     if ( SUCCESS == error_code ) error_code = ( (int) qty_objects );
1708     return error_code;
1709 }
```

4.1.4.82 int SPI_FIFO_Write (const void * *buffer*, size_t *size*, size_t *count*, FILE * *fd_log*)

Writes specifically to the SPI transmit/write data FIFO. It does not attempt to correlate the number of transmit bytes with receive bytes. Its purpose is more for low level hardware testing. Note that this function has a signature identical to the fwrite() function (i.e. make use of function pointers to guide destination of data).

Parameters

in	*buffer*	Buffer containing the data to be written.

in	*size*	Size of objects in bytes.
in	*count*	Number of objects to be written
out	*fd_log*	Optional log file to write the buffer too. If NULL, then no logging.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned indicating an error. In addition, a positive value is returned indicating the number of actual objects written.

Definition at line 1628 of file idi.c.

References IO_Write_U8(), SPI_FIFO_SIZE, SPI_IsNotPresent(), and SPI_Status_Write_FIFO_Status().

Referenced by IDI_CMD__SPI_FIFO().

```
1629 {
1630     int     error_code;       /* used primarily for debug purposes */
1631     size_t  bytes_in_fifo;
1632     BOOL    empty;
1633     BOOL    full;
1634     size_t  index;
1635     size_t  qty_objects;
1636     size_t  qty_bytes;
1637
1638     error_code = SUCCESS;
1639
1640     if ( (size * count) > SPI_FIFO_SIZE ) return -EC_PARAMETER;
1641
1642     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
1643
1644     SPI_Status_Write_FIFO_Status( &full, &empty, &bytes_in_fifo );
1645
1646     qty_objects =  (SPI_FIFO_SIZE - bytes_in_fifo) / size; /* max number of objects that can
      be processed */
1647     if ( count > qty_objects ) count = qty_objects;
1648
1649     qty_bytes = count * size;
1650
1651     for ( index = 0; index < qty_bytes; index++ ) IO_Write_U8( SPI_DATA, ((uint8_t *) buffer)[
     index] );
1652
1653     if ( NULL != fd_log )
1654     {
1655         error_code = fwrite( buffer, size, qty_objects, fd_log );
1656     }
1657
1658     if ( SUCCESS == error_code ) error_code = ( (int) qty_objects );
1659     return error_code;
1660 }
```

4.1.4.83 int SPI_ID_Get (uint16_t * *id*)

Retrieves the SPI ID register value.

Parameters

out	*id*	The SPI component ID value.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1041 of file idi.c.

References ID_SPI, and IO_Read_U8().

Referenced by IDI_CMD__SPI_ID(), and SPI_IsNotPresent().

```
1042 {
1043     uint8_t     lsb, msb;
1044
1045     IO_Read_U8( SPI_ID_LSB, &lsb );
1046     IO_Read_U8( SPI_ID_MSB, &msb );
1047     *id = ( ((uint16_t) msb) << 8 ) | ((uint16_t) lsb);
1048 #if defined( ID_ALWAYS_REPORT_AS_GOOD )

1049     *id = ID_SPI;
1050 #endif

1051     return SUCCESS;
1052 }
```

4.1.4.84 int SPI_IsNotPresent (void)

Reports if the SPI component is available within the register space by matching a known ID. The SPI register map is only enabled within the hardware if the hardware mode is not zero (i.e. M1 and M0 jumpers on the board provide a nonzero value).

Returns

A zero is returned if the SPI component ID is not found within the register space.

Definition at line 1062 of file idi.c.

References ID_SPI, and SPI_ID_Get().

Referenced by SPI_Commit(), SPI_Configuration_Chip_Select_Behavior_Get(), SPI_Configuration_Chip_Select_Behavior_Set(), SPI_Configuration_Get(), SPI_Configuration_Set(), SPI_Data_Write_Read(), SPI_FIFO_Read(), SPI_FIFO_Write(), SPI_Status_Read(), and SPI_Status_Write().

```
1063 {
1064     uint16_t id;
1065     SPI_ID_Get( &id );
1066     if (ID_SPI == id ) return 0;
1067     return 1;
1068 }
```

4.1.4.85 int SPI_Report_Configuration_Text (struct spi_cfg * *cfg*, FILE * *out*)

Creates a human readable report of the SPI configuration data structure.

Parameters

in	*cfg*	SPI configuration data structure pointer [out] out File destination descriptor

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1346 of file idi.c.

References spi_cfg::chip_select_behavior, spi_cfg::clock_hz, spi_cfg::end_cycle_delay, spi_cfg::end_delay_ns, spi_cfg::half_clock_interval, IDI_CSB_BUFFER, IDI_CSB_SOFTWARE, IDI_CSB_UINT16, IDI_CSB_UINT8, spi_cfg::sclk_phase, spi_cfg::sclk_polarity, and spi_cfg::sdi_polarity.

Referenced by IDI_CMD__SPI_Config_Get().

```
1347 {
1348     fprintf( out, "################################## SPI Configuration:\n" );
1349     fprintf( out, "sdio_wrap             = %s\n", cfg->sclk_polarity   ? "true" : "false" );
1350     fprintf( out, "sdo_polarity          = %s\n", cfg->sclk_polarity   ? "true" : "false" );
1351     fprintf( out, "sdi_polarity          = %s\n", cfg->sdi_polarity     ? "true" : "false" );
1352     fprintf( out, "sclk_phase            = %s\n", cfg->sclk_phase         ? "true" : "false" );
1353     fprintf( out, "sclk_polarity         = %s\n", cfg->sclk_polarity   ? "true" : "false" );
1354     fprintf( out, "chip_select_behavior  = " );
1355
1356     switch( cfg->chip_select_behavior )
1357     {
1358         case IDI_CSB_SOFTWARE:      fprintf( out, "IDI_CSB_SOFTWARE" );         break;
1359         case IDI_CSB_BUFFER:      fprintf( out, "IDI_CSB_BUFFER" );           break;
1360         case IDI_CSB_UINT8:           fprintf( out, "IDI_CSB_UINT8" );            break;
1361         case IDI_CSB_UINT16:      fprintf( out, "IDI_CSB_UINT16" );           break;
1362         default:                    fprintf( out, "undefined" );                break;
1363     }
1364     fprintf( out, "\n" );
1365
1366     fprintf( out, "end_cycle_delay       = 0x%02X  (%d)\n", cfg->end_cycle_delay,     cfg->
     end_cycle_delay     );
1367     fprintf( out, "half_clock_interval   = 0x%04X  (%d)\n", cfg->
     half_clock_interval, cfg->half_clock_interval );
1368
1369     fprintf( out, "clock_hz              = %f Hz\n", cfg->clock_hz );
1370     fprintf( out, "end_delay_ns          = %f ns\n", cfg->end_delay_ns );
1371     fprintf( out, "\n" );
1372     return SUCCESS;
1373 }
```

4.1.4.86 int SPI_Report_Status_Text (struct spi_status * *status*, FILE * *out*)

Produces a human readable report of the SPI status data structure.

Parameters

in	*status*	SPI status data structure pointer [out] out File destination descriptor

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1384 of file idi.c.

References spi_status::empty, spi_status::fifo_count, spi_status::fifo_size, spi_status::full, and spi_status::tx_status.

Referenced by IDI_CMD__SPI_Status().

```
1385 {
1386     fprintf( out, "################################## SPI " );
1387     if ( status->tx_status ) fprintf( out, "TX" );
1388     else                     fprintf( out, "RX" );
1389
1390     fprintf( out, " Status:\n" );
1391
1392     fprintf( out, "full       = %s\n", status->full  ? "true" : "false" );
1393     fprintf( out, "empty      = %s\n", status->empty ? "true" : "false" );
1394     fprintf( out, "fifo size  = %d\n", status->fifo_size                );
1395     fprintf( out, "fifo count = %d\n", status->fifo_count               );
1396
1397     return SUCCESS;
1398 }
```

4.1.4.87 int SPI_Status_Read (struct spi_status * *status*)

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

Parameters

out	*status*	Pointer to status data structure to be updated.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1550 of file idi.c.

References spi_status::empty, spi_status::fifo_count, spi_status::fifo_size, spi_status::full, IO_Read_U8(), SPI_FIFO_SIZE, SPI_IsNotPresent(), and spi_status::tx_status.

Referenced by IDI_CMD__SPI_Status().

```
1551 {
1552     uint8_t reg_value;
1553
1554     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
1555
1556     IO_Read_U8( SPI_RX_STATUS, &reg_value );
1557     status->fifo_count = (int)( reg_value & 0x1F );
1558     status->full       = (BOOL)( reg_value & 0x80 );
1559     status->fifo_size  = (int) SPI_FIFO_SIZE;
1560     status->empty      = (BOOL)( reg_value & 0x40 );
1561     status->tx_status  = false;
1562     return SUCCESS;
1563 }
```

4.1.4.88 BOOL SPI_Status_Read_FIFO_Is_Not_Empty (void)

Returns the receive/read FIFO empty status flag. This function is typically used to determine if the FIFO is empty.

Returns

Returns true if the receive/read FIFO is not empty.

Definition at line 1587 of file idi.c.

References IO_Read_U8().

Referenced by IDI_CMD__SPI_FIFO().

```
1588 {
1589     uint8_t reg_value;
1590
1591     IO_Read_U8( SPI_RX_STATUS, &reg_value );
1592     if ( reg_value & 0x40 ) return false;
1593     return true;
1594 }
```

4.1.4.89 void SPI_Status_Read_FIFO_Status (BOOL * *empty*, size_t * *bytes_available*)

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

Parameters

out	*empty*	FIFO empty flag
out	*bytes_available*	a count of the number of bytes in the FIFO

Returns

nothing

Definition at line 1524 of file idi.c.

References IO_Read_U8().

Referenced by SPI_Data_Write_Read_Helper(), and SPI_FIFO_Read().

```
1525 {
1526     uint8_t reg_value;
1527
1528     IO_Read_U8( SPI_RX_STATUS, &reg_value );
1529     *bytes_available = (size_t)( reg_value & 0x1F );
1530     *empty           = (BOOL)( reg_value & 0x40 );
1531 }
```

4.1.4.90 int SPI_Status_Write (struct spi_status * *status*)

Builds a detailed status data structure of the transmit/write outgoing SPI data FIFO. Reports the quantity of bytes currently in the transmit FIFO, full flag, empty flag, the total size of the FIFO in bytes, and sets tx_status to true indicating that this is status specific to the transmit 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

Parameters

out	*status*	Pointer to status data structure to be updated.

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 1449 of file idi.c.

References spi_status::empty, spi_status::fifo_count, spi_status::fifo_size, spi_status::full, IO_Read_U8(), SPI_FIFO_↩ SIZE, SPI_IsNotPresent(), and spi_status::tx_status.

Referenced by IDI_CMD__SPI_Status().

```
1450 {
1451     uint8_t reg_value;
1452
1453     if ( SPI_IsNotPresent() ) return -EC_SPI_NOT_FOUND;
1454
1455     IO_Read_U8( SPI_TX_STATUS, &reg_value );
1456     status->fifo_count = (int)( reg_value & 0x1F );
1457     status->full       = (BOOL)( reg_value & 0x80 );
1458     status->fifo_size  = (int) SPI_FIFO_SIZE;
1459     status->empty      = (BOOL)( reg_value & 0x40 );
1460     status->tx_status  = true;
1461     return SUCCESS;
1462 }
```

4.1.4.91 BOOL SPI_Status_Write_FIFO_Is_Full (void)

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.

Returns

Returns true if the transmit/write FIFO is full.

Definition at line 1501 of file idi.c.

References IO_Read_U8().

```
1502 {
1503     uint8_t reg_value;
1504     IO_Read_U8( SPI_TX_STATUS, &reg_value );
1505     if ( reg_value & 0x80 ) return true;
1506     return false;
1507 }
```

4.1.4.92 BOOL SPI_Status_Write_FIFO_Is_Not_Empty (void)

Returns the transmit/write FIFO empty status flag. This function is typically used to wait for the transmit/write FIFO to become empty.

Returns

Returns true if the transmit/write FIFO is not empty.

Definition at line 1572 of file idi.c.

References IO_Read_U8().

Referenced by FRAM__Memory_Read(), FRAM__Memory_Write(), and IDI_CMD__SPI_FIFO().

```
1573 {
1574     uint8_t reg_value;
1575     IO_Read_U8( SPI_TX_STATUS, &reg_value );
1576     if ( reg_value & 0x40 ) return false;
1577     return true;
1578 }
```

4.1.4.93 void SPI_Status_Write_FIFO_Status (BOOL * *full*, BOOL * *empty*, size_t * *bytes_in_fifo*)

Returns the complete write/transmit 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

Parameters

out	*full*	FIFO full flag
out	*empty*	FIFO empty flag

out	*bytes_in_fifo*	a count of the number of bytes in the FIFO

Returns

nothing

Definition at line 1480 of file idi.c.

References IO_Read_U8(), and SPI_FIFO_SIZE.

Referenced by SPI_Data_Write_Read_Helper(), and SPI_FIFO_Write().

```
1481 {
1482     uint8_t reg_value;
1483     IO_Read_U8( SPI_TX_STATUS, &reg_value );
1484     switch( reg_value & 0xC0 )
1485     {
1486         case 0x00: *full = false;   *empty = false;        break;
1487         case 0x40: *full = false;   *empty = true;     break;
1488         case 0x80: *full = true;    *empty = false;        break;
1489         case 0xC0: *full = true;    *empty = true;     break;
1490     }
1491     *bytes_in_fifo = (size_t) SPI_FIFO_SIZE - (size_t)( reg_value & 0x1F );
1492 }
```

4.1.4.94 BOOL String_To_Bool (const char * *str*)

General function used to convert a string into a boolean equivalent value.

Parameters

in	*str*	string input for conversion.

Returns

a BOOL is returned. The default value returned is false.

Definition at line 541 of file idi.c.

Referenced by IDI_CMD__Main_IO_Behavior(), IDI_CMD__SPI_Commit(), IDI_CMD__SPI_Config_SDI_Polarity(), IDI_CMD__SPI_Config_SDIO_Wrap(), IDI_CMD__SPI_Config_SDO_Polarity(), and IDI_CMD__SPI_FIFO().

```
542 {
543     switch( str[0] )
544     {
545     case '0':
546     case 'f':
547     case 'F':
548         return false;
549     case '1':
550     case 't':
551     case 'T':
552         return true;
553     }
554     return false;
555 }
```

4.1.5 Variable Documentation

4.1.5.1 const struct reg_definition definitions[] [static]

Initial value:

```
=
{

}
```

Definition at line 433 of file idi.c.

4.1.5.2 const struct ec_human_readable ec_human_readable[ ]

Initial value:

```
=
{

        EC_HUMAN_READABLE_TERMINATE
}
```

Definition at line 411 of file idi.c.

4.1.5.3 const char ec_unknown[ ] = "unknown error code" [static]

Translates an error code into a human readable message.

Parameters

in	*error_code*	The error code to be translated into a human readable message

Returns

a human readable string representing a very brief description of the error code.

Definition at line 624 of file idi.c.

Referenced by EC_Code_To_Human_Readable().

4.1.5.4 const char* idi_bank_symbol_names[ ] [static]

Initial value:

```
=
{
    "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"
}
```

Definition at line 418 of file idi.c.

Referenced by IDI_Symbol_Name_Bank().

4.1.5.5 struct command_line idi_cmd_din[] [static]

Initial value:

```
=
{
    {   IDI_CMD__DIN_ID,      "id",       "params: none.  Reports the DIN board/component ID.
      "    },
    {   IDI_CMD__DIN_Channel,   "chan",     "params: <channel>"
      },
    {   IDI_CMD__DIN_Group,       "group",    "params: [<group_channel | all>]"
      },
    {   IDI_CMD__DIN_All,       "all",      "reports all digital inputs in binary and hex"
      },
    {   NULL,                   NULL,       NULL                                                },
}
```

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3411 of file idi.c.

4.1.5.6 struct command_line idi_cmd_fram[] [static]

Initial value:

```
=
{
    {   IDI_CMD__FRAM_Dump,   "dump",     "params: <address> <length>"
      },
    {   IDI_CMD__FRAM_Save,       "save"      "params: <address> <length> <binary
       destination file>"  },
    {   IDI_CMD__FRAM_Load,       "load"      "params: <address> <binary source file name
      "           },
    {   IDI_CMD__FRAM_Init,       "init"      "params: [byte/character] [byte/character]
       ..."         },
    {   IDI_CMD__FRAM_WREN,       "wren"      "WRite Enable Latch Set"
      },
    {   IDI_CMD__FRAM_WRDI,       "wrdi"      "WRite DIsable"
      },
    {   IDI_CMD__FRAM_RDSR,       "rdsr"      "ReaD Status Register"
      },
    {   IDI_CMD__FRAM_WRSR,       "wrsr"      "WRite Status Register.  Params: <status>"
      },
    {   IDI_CMD__FRAM_RDID,       "rdid"      "ReaD ID Register"
      },
    {   NULL,                   NULL,       NULL                                                },
}
```

Definition at line 3213 of file idi.c.

4.1.5.7 struct command_line idi_cmd_main[] [static]

Initial value:

```
=
{
    {   IDI_CMD__Main_Base,               "base",     "params: [<address>]"
      },
    {   IDI_CMD__Main_IO_Behavior,     "io",       "params: [<simulate>/<report>]"
      },
    {   NULL,                           NULL,       NULL
      },
}
```

Returns

A zero (SUCCESS) is returned if successful, otherwise a negative error code is returned.

Definition at line 3604 of file idi.c.

4.1.5.8 struct command_line idi_cmd_spi[ ] [static]

Initial value:

```
=
{
    {   IDI_CMD__SPI_ID,                              "id",       "wishbone id: params: none"
      },
    {   IDI_CMD__SPI_Config_Get,                      "cfg",      "config dump:
     params: none"                                 },
    {   IDI_CMD__SPI_Config_Clock_Hz,                 "clk",      "clk:
     params: [<clock freq in hertz>]"              },
    {   IDI_CMD__SPI_Config_End_Cycle_Delay_Sec, "ecd",      "end
     delay:   params: [<time in seconds>]"                 },
    {   IDI_CMD__SPI_Config_Mode,                     "mode",     "mode:
     params: [<0/1/2/3>]"                          },
    {   IDI_CMD__SPI_Config_SDI_Polarity,             "sdi",      "sdi pol:
      params: [<true/1/false/0>]"                   },
    {   IDI_CMD__SPI_Config_SDO_Polarity,             "sdo",      "sdo pol:
      params: [<true/1/false/0>]"                   },
    {   IDI_CMD__SPI_Config_SDIO_Wrap,                "wrap",     "sdo-->sdi:
     params: [<true/1/false/0>]"                   },
    {   IDI_CMD__SPI_Config_Chip_Select_Behavior,     "csb",      "
    chip select behavior: params: [0/1/2/3/software/buffer/uint8/uint16]"  },
    {   IDI_CMD__SPI_Status,                       "status",   "status of both TX and RX
     buffers"                          },
    {   IDI_CMD__SPI_Data,                           "data",     "read/write:  params: [one
     or more bytes/characters]"    },
    {   IDI_CMD__SPI_FIFO,                           "fifo",     "fifo r/w:    params: [one
     or more bytes/characters]"    },
    {   IDI_CMD__SPI_Commit,                       "commit"    "causes spi transactions to
     start"                         },
    {   NULL,                                    NULL,       NULL
      }
}
```

Definition at line 2994 of file idi.c.

4.1.5.9 struct idi_dataset idi_dataset

Definition at line 441 of file idi.c.

4.1.5.10 const char idi_svn_revision_string[ ] = { IDI_REV } [static]

Global variables.

Definition at line 408 of file idi.c.

Referenced by IDI_Initialization().